INTRODUCTION

This book is your definitive guide to understanding Linux and Ubuntu at every level - from the moment you first install it to architecting complex systems that run the modern internet. This isn't a tutorial. This isn't a reference manual. This is the complete curriculum that gatekeeping institutions protect, now available to anyone who wants to understand operating systems deeply.

Why Ubuntu? Because millions of people start their Linux journey with Ubuntu. It's the most accessible entry point to the Linux ecosystem, and understanding it means understanding all of Linux. More importantly, every principle you learn here - from kernel concepts to system administration - applies to every other Linux distribution and informs how you understand operating systems in general.

This book answers the six fundamental questions about Linux and Ubuntu:

What - What exactly is Ubuntu? What are its components? How does it break down?
Who - Who created it? Who uses it? What roles benefit from this knowledge?
Whom - Who does this affect? What problems does it solve? Who needs this?
How - How does it actually work internally? How do pieces interact?
When - When was it created? When do you use different components? Historical context?
Why - Why does it exist? Why was it designed this way? Why do alternatives exist?

Every concept in this book answers all six of these questions. This ensures you understand not just how things work, but why they were designed that way, who benefits, and what alternatives exist.

SECTION 1: FOUNDATIONS - WHAT IS LINUX AND UBUNTU ANYWAY?

The Story and the Why

Linux began in 1991 as a kernel - just the core scheduling and memory management system - written by Linus Torvalds as a hobby project on his Intel 386 PC. He didn't know he was creating something that would eventually power the majority of internet servers, smartphones, and embedded devices worldwide.

A kernel alone doesn't make an operating system. You need shells to give users a way to interact with the system. You need utilities and tools to do actual work. You need libraries so programmers don't reinvent everything. You need package managers so people can install software. You need init systems to start services on boot. Linux the kernel is like an engine - powerful but useless without a car built around it.

This is where GNU comes in. The GNU project (which stands for "GNU's Not Unix" - a recursive acronym that programmers love) created the utilities, libraries, and tools. The combination of the Linux kernel with GNU tools created what we call GNU/Linux. Most people just call it Linux for short, and we'll do the same here.

Ubuntu takes this combination and packages it with beautiful interfaces, sensible defaults, and a community focused on making Linux accessible to humans. The name comes from the Zulu and Xhosa words meaning "humanity toward others" - reflecting the philosophy of the project. Canonical, the company behind Ubuntu, maintains it and makes money by offering support and services rather than licensing fees.

This is why Ubuntu matters: it proved Linux could be user-friendly. Before Ubuntu in 2004, Linux was largely the domain of server administrators and programmers. Ubuntu showed that ordinary people could use it for everyday computing. That's powerful.

Historical Context: Why Linux Exists At All

Before Linux, if you wanted an operating system, you either bought expensive commercial systems like Solaris or AIX, or you used personal computer operating systems like DOS or early Windows that weren't designed for servers or serious computing.

Unix was the gold standard for serious computing, but it was proprietary. You couldn't see the source code. You couldn't modify it. You couldn't build on it freely.

Richard Stallman and the free software movement believed software should be free - not free as in price, but free as in freedom. The GNU project's goal was to create a Unix-compatible system that was completely free and open.

The missing piece was the kernel. Minix was a teaching kernel, but Linus Torvalds created something better. When he released Linux under the GPL (General Public License), it meant anyone could use it, modify it, and redistribute it as long as they shared the source code. This legal framework enabled the collaborative development that built modern

Linux.

Ubuntu came later with a mission: make this powerful but intimidating technology accessible to regular humans. Ubuntu almost single-handedly made Linux relevant on desktops and laptops, not just servers.

The Why Behind the Design Choices

When you understand the history, you understand the design. Linux is Unix-like because Unix proved a good model for systems design. The kernel handles the lowest-level hardware management so applications don't need to understand every possible hardware variant. The shell provides a human-readable interface. Tools are designed to do one thing well and work together through pipes and text.

This came from experience. The Unix philosophy emerged from decades of practical systems programming. It wasn't theoretical. It was "here's what actually works when you're building systems that need to be reliable, maintainable, and extensible."

Ubuntu's design priorities reflect modern computing. When Ubuntu launched, mobile devices, cloud computing, and virtualization were coming. Ubuntu was designed to scale from a laptop to thousands of servers seamlessly. The same system concepts work whether you're running one instance or one million.

What Is Linux Actually?

At its core, Linux is a kernel. The kernel is the program that:

1. Manages the CPU - decides which processes get to run when

2. Manages memory - keeps applications from stepping on each other
3. Manages storage - provides the filesystem interface
4. Manages devices - drives for keyboards, mice, networks, disks
5. Manages security - enforces permissions and protections

That's genuinely it. Everything else is tools and applications built on top of the kernel's abstractions.

What Is Ubuntu?

Ubuntu is Linux (the kernel) plus:

GNU utilities and tools
A package manager
A system initialization framework
Desktop environments (for the desktop version) or just services (for the server)
Sensible default configurations
Community support

Ubuntu comes in flavors. Ubuntu Desktop is for personal computers with a graphical interface. Ubuntu Server is for servers without the graphical desktop. Other flavors like Kubuntu, Xubuntu, Lubuntu use different desktop environments but share the same core.

When you install Ubuntu, you're getting:

- The Linux kernel managing hardware
- GNU tools providing core functionality
- Apt package manager for installing software
- Systemd for starting services on boot
- Bash shell for command-line interaction
- Thousands of pre-built applications

All of this works together as a cohesive system.

SECTION 2: THE MENTAL MODEL - HOW TO THINK ABOUT LINUX

Everything Is A File

The Unix philosophy embedded in Linux is radical in its simplicity: everything is a file.

Your actual data files - documents, music, code - these are obviously files.

But in Linux, your hard drive is also represented as a file: /dev/sda

Your network interface is a file: /dev/eth0

A process in memory can be viewed as a file in /proc

Your terminal is a file: /dev/tty

Even devices that don't store data - like your network card or USB port - are represented as files that you interact with.

This abstraction is powerful. You learned how to read a file, write to a file, and change file permissions. Those same operations work on devices, network interfaces, and processes. A programmer can open what looks like a file and not know whether they're reading from disk, from a network socket, or from a device driver. The filesystem provides a unified interface.

The Filesystem Hierarchy

Everything in Linux starts from root. The root filesystem is represented as / (forward slash). Everything - every file, every directory, every device - is somewhere under this root.

When you see a path like /home/hashimo/Documents/file.txt, the slashes mean "in the directory... in the directory... in the directory." This path means: in root, there's a directory called home, in home there's a directory called hashimo, in hashimo there's a directory called Documents, and in Documents there's a file called file.txt.

This tree structure organizes the entire system. Ubuntu follows the Linux Filesystem Hierarchy Standard which defines what goes where:

/bin - Essential command binaries
/boot - Boot loader and kernel files
/dev - Device files
/etc - Configuration files
/home - User home directories
/lib - System libraries
/media - Removable media mount points
/mnt - Temporary mount points
/opt - Optional software packages
/proc - Process information
/root - Root user's home directory
/run - Runtime data
/sbin - System administration binaries
/srv - Service data
/sys - System information
/tmp - Temporary files
/usr - User programs and libraries
/var - Variable data like logs

Understanding this hierarchy is crucial. When you're looking for something, you know where to look. Configuration files go

in /etc. User data goes in /home. System libraries go in /lib. Logs go in /var/log.

Processes and The Kernel

A process is a running program. When you double-click an application or type a command, you're creating a process. The kernel's job is to manage all these processes fairly.

The kernel uses something called scheduling to decide which process gets CPU time. It doesn't matter if you have one CPU - the kernel can make it feel like multiple processes are running simultaneously by switching between them. If you have multiple CPUs, it can truly run multiple processes in parallel.

Each process gets:

- Its own memory space (isolated from other processes)
- A unique process ID (PID)
- Environment variables it can access
- File descriptors for files it can read/write
- Permissions based on which user ran it

One crucial concept: the kernel is the only program that can directly access hardware. Applications run in user space. They can't directly access disk drives or network cards. They ask the kernel. This is how security works. The kernel can say "no, you don't have permission for that."

Users and Permissions

Linux is fundamentally a multi-user system. Multiple users can be logged in simultaneously, each with their own processes, files, and environment.

Each file has permissions:

- Owner (usually the user who created it)
- Group (a collection of users)

- Everyone else

For each of these three categories, you can specify:

- Read permission (can view the file)
- Write permission (can modify the file)
- Execute permission (can run the file if it's a program)

When a user tries to access a file, the kernel checks: is this the owner (check owner permissions), is this user in the group (check group permissions), or neither (check everyone permissions). If the relevant permission is granted, the access is allowed. If not, it's denied.

The root user is special. Root can do anything - access any file, run any command, install any software. This is why you must be careful with root access. A malicious command run as root can destroy your entire system.

This model of users and permissions is how multiple people can share a computer safely. My home directory is mine. I can read my own files. I can't read your files unless you gave me permission. The system enforces this automatically.

SECTION 3: GETTING STARTED - INSTALLING AND FIRST STEPS

Installing Ubuntu

Ubuntu installation is now quite straightforward. You create a bootable USB drive with the Ubuntu installer, restart your computer, and follow the prompts. The installer will:

1. Detect your hardware

2. Ask where you want to install (careful here - you can overwrite existing installations)
3. Ask for your user account information
4. Configure your network
5. Download updates and core packages
6. Finish by rebooting into your new Ubuntu system

The installer is deliberately simple. It hides complexity. Underneath it's configuring the bootloader, partitioning the disk, installing thousands of packages, and setting up your system. But you don't see that complexity. You just see a few screens with reasonable defaults.

This is important design philosophy in Ubuntu. Make common cases simple. Power users can still access advanced options, but the default path is accessible.

After Installation: First Boot

When you boot into Ubuntu for the first time, you see a login screen. You enter the username and password you created during installation. The login program verifies your credentials, creates an environment for your session, and starts your shell.

If you're using the desktop version, a graphical desktop environment loads - probably GNOME. This provides windows, menus, icons, and mouse-based interaction. Under the hood, GNOME is thousands of small programs working together - a window manager that positions windows, a file manager that displays folders, a panel that shows time and system status.

If you're using the server version, you get a command-line login. No graphical interface. You type commands, and the system responds with text.

Both approaches are doing the same thing at the kernel level. The kernel manages hardware, memory, and processes. The difference is what runs on top.

The Shell and Command Line

The shell is a program that interprets your commands. In Ubuntu, the default shell is Bash (Bourne Again Shell). The shell's job is simple: wait for you to type a command, parse what you typed, find the program, run it with the arguments you provided, and show you the output.

When you open a terminal in the desktop environment or log in on the server, you're starting a shell. You see a prompt (usually ending with $ for regular users or # for root). You type a command. You press enter. The shell runs it.

For example, you might type:

```
ls -la /home
```

This tells the shell: run the program called ls with arguments -la and /home. The ls program lists files. The -la arguments mean show all files including hidden ones (starting with dot) in long format with details. The /home argument means list the directory /home.

The shell finds the ls program (usually in /bin/ls), starts it with those arguments, waits for it to complete, and displays the output. You see something like:

```
drwxr-xr-x  3 root root  4096 Nov 28 10:30 .
drwxr-xr-x 12 root root  4096 Nov 28 10:30 ..
drwxr-xr-x 30 hashimo hashimo 4096 Dec 11 16:00 hashimo
```

These lines show directories and their permissions. The first column shows permissions. The next shows link count. Then

owner and group. Then size. Then date. Then name.

Learning to read and interpret this output is foundational. Everything you do in Linux goes through this command-line interface.

Basic Commands You Need To Know

pwd - Print working directory. Shows you which directory you're currently in.

cd - Change directory. Moves you to a different directory.

ls - List directory contents. Shows what's in the current directory.

mkdir - Make directory. Creates a new folder.

rm - Remove. Deletes files or directories.

cp - Copy. Copies files or directories.

mv - Move. Moves or renames files or directories.

cat - Concatenate. Displays file contents.

grep - Search. Finds text patterns in files.

sudo - Superuser do. Runs a command as the root user.

These are not the only commands. There are thousands. But these are the core ones you'll use daily. Everything else builds on top of these concepts.

Here's a practical example. You want to create a folder for your project, copy some files into it, and see what's there:

```
mkdir my_project
```

This creates a folder called my_project in your current

directory.

cd my_project

This moves into that directory. Now your working directory is my_project.

cp ~/Documents/file1.txt .

This copies a file from your Documents folder into the current directory. The . means "current directory."

cp ~/Documents/file2.txt .

Copy another file.

ls

List what's in this directory. You see your two files.

pwd

Print your working directory. You see something like /home/hashimo/my_project.

This is the basic workflow. Navigate directories, create files, copy them, organize them. Everything in Linux is this: managing the filesystem through commands.

SECTION 4: PACKAGE MANAGEMENT - HOW UBUNTU INSTALLS SOFTWARE

The Problem: Where Does Software Come From?

In the early days of computers, you'd download a program, double-click it, and hope it worked. But installing software is

complicated:

The program might depend on libraries you don't have. It might need specific versions of those libraries. It might conflict with other software you already installed. If you want to uninstall it, you need to remove the program and its dependencies, but not if other programs also depend on those libraries.

Linux solved this with package managers. A package is a compressed archive containing a program, its dependencies, and installation instructions. The package manager handles:

Finding the right package
Checking dependencies (what other packages this one needs)
Downloading it
Installing it correctly
Managing dependencies
Allowing you to remove it cleanly

Ubuntu uses APT (Advanced Package Tool). There's a lower level called dpkg that actually handles installation, but APT sits on top and makes it user-friendly.

How APT Works

First, you update your package lists. APT knows where to find packages - repositories. The default repositories are Canonical's official servers, but you can add others. When you update, APT downloads the current list of available packages and their versions.

```
sudo apt update
```

This command updates your package lists. It connects to the repositories, checks what packages are available, and stores the information locally. You need sudo because modifying the

system requires root access.

Then you can search for packages:

```
apt search nginx
```

This searches available packages for anything with "nginx" in the name or description. You see results like:

nginx - small, powerful, scalable web server and reverse proxy
nginx-core - small, powerful, scalable web server and reverse proxy (core package)
nginx-extras - small, powerful, scalable web server and reverse proxy (extras)

If you want to install one:

```
sudo apt install nginx
```

APT figures out what nginx needs. Maybe it depends on openssl and zlib. APT checks - do you have those? If not, it adds them to the installation list. It shows you what it's going to install and asks for confirmation. You press y and hit enter. APT downloads everything and installs it in the right places. Your system now has a working web server.

If you want to uninstall it:

```
sudo apt remove nginx
```

APT removes nginx. It leaves libraries like openssl and zlib because other packages might depend on them. If you want to remove unused dependencies:

```
sudo apt autoremove
```

Breaking down:

sudo apt - Package manager
autoremove - Remove unused packages. APT removes packages that nothing depends on anymore

Completely remove with config files:

sudo apt purge nginx

Breaking down:

sudo apt purge - Remove package completely
nginx - Package name

The difference between remove and purge:

apt remove - Removes the package but leaves configuration files
apt purge - Removes the package AND all configuration files

When to use each:

Remove: You might reinstall and want to keep settings
Purge: You want to completely get rid of package (cleaner system)

After removing, clean up unused:

sudo apt autoremove
sudo apt autoclean

Breaking down:

autoremove - Remove unused dependency packages
autoclean - Remove cached package files

If you want to update your software:

sudo apt update
sudo apt upgrade

The first command updates the package lists and sees that newer versions are available. The second command upgrades everything to the newest versions.

This is the power of package management. You don't manually download anything. You don't worry about dependencies. You don't manage files scattered across your system. The package manager handles all of it.

Repositories and Trust

APT gets packages from repositories - servers that store packages. Canonical maintains official repositories. But anyone can set up a repository. You can add additional repositories to get software not in the official ones.

When you add a repository, you usually also add a GPG key. This key verifies that packages are authentic and haven't been tampered with. Without this verification, someone could set up a fake repository, serve you malicious packages, and you'd install them not realizing they're not trustworthy.

This is how Linux is secure. You trust Canonical. Canonical maintains the repositories and verifies packages. You trust the GPG keys. Everything you install from the official repositories is checked and verified.

SECTION 5: REAL-WORLD PRACTICAL EXAMPLES - FROM THEORY TO ACTUAL WORK

Scenario 1: Setting Up a Web Development Environment

Let's say you're a web developer. You want to set up your laptop to build websites. You need a database, a web server, and a text editor. Here's exactly how you do it in Ubuntu.

First, open a terminal. You should see something like this:

hashimo@ubuntu:~$

This tells you:

- hashimo is your username
- ubuntu is your computer name
- ~ means you're in your home directory
- $ means you're a regular user (not root)

Your first step is updating your package lists. You type:

$ sudo apt update

You'll see output like:

Get:1 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Hit:3 http://archive.ubuntu.com/ubuntu focal-backports InRelease
Reading package lists... Done

This is APT connecting to the repositories and downloading

the list of available packages. When it's done, you're ready to install software.

Now install your tools. First, a code editor:

```
$ sudo apt install code
```

APT shows you what it's installing:

The following NEW packages will be installed:
code
Processing triggers for mime-support (3.64ubuntu1) ...
Processing triggers for desktop-file-utils (0.26-1ubuntu1) ...

Done. Visual Studio Code is installed.

Next, install a web server:

```
$ sudo apt install nginx
```

Then a database:

```
$ sudo apt install postgresql postgresql-contrib
```

When it's done, check what you have:

```
$ which code
/usr/bin/code

$ which nginx
/usr/sbin/nginx

$ which psql
/usr/bin/psql
```

These commands show you where each program is installed. They're in the system directories where APT put them. Your system can find them because those directories are in your PATH - a special variable that tells the shell where to look for programs.

Now you want to see what's running. Is PostgreSQL actually started?

$ systemctl status postgresql

This shows you:

● postgresql.service - PostgreSQL RDBMS
Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)
Active: active (exited) since Thu 2025-12-11 17:00:22 UTC; 1 min 23s ago
Process: 2841 ExecStart=/usr/lib/postgresql/start-postgresql

Active means PostgreSQL is running. The service is managing it automatically - it started when your system booted.

Scenario 2: Organizing Your Project Files

Now you have your tools. You want to organize a project. You decide to create a structure for a web application. Here's what you do:

$ mkdir -p ~/projects/myapp/{backend,frontend,database,docs}

This creates a directory structure in one command:

```
myapp/
 backend/
 frontend/
 database/
 docs/
```

The -p flag means "create parent directories if they don't exist." The curly braces let you create multiple directories at once.

Now navigate to your project:

```
$ cd ~/projects/myapp
```

Check where you are:

```
$ pwd
/home/hashimo/projects/myapp
```

List what's in there:

```
$ ls -la
```

This shows:

```
drwxr-xr-x  6 hashimo hashimo 4096 Dec 11 17:05 .
drwxr-xr-x  3 hashimo hashimo 4096 Dec 11 17:04 ..
drwxr-xr-x  2 hashimo hashimo 4096 Dec 11 17:05 backend
drwxr-xr-x  2 hashimo hashimo 4096 Dec 11 17:05 database
drwxr-xr-x  2 hashimo hashimo 4096 Dec 11 17:05 docs
drwxr-xr-x  2 hashimo hashimo 4096 Dec 11 17:05 frontend
```

Perfect. Four directories ready for your work.

Scenario 3: Creating and Editing Files

You want to create a README explaining your

project. You use a text editor:

```
$ nano README.md
```

This opens Nano, a simple text editor. You type:

```
# My Web Application

This is a project for learning web development with Ubuntu.

## Structure
- backend: Python Flask API
- frontend: React application
- database: PostgreSQL schemas
- docs: Documentation
```

You save (Ctrl + O, press Enter) and exit (Ctrl + X).

Now check your file exists:

```
$ ls -la README.md

-rw-r--r-- 1 hashimo hashimo 142 Dec 11 17:07 README.md
```

The first part (-rw-r--r--) shows permissions: the owner (you) can read and write, the group can only read, everyone can only read.

View the file:

```
$ cat README.md

# My Web Application

This is a project for learning web development with Ubuntu.

## Structure
- backend: Python Flask API
```

- frontend: React application
- database: PostgreSQL schemas
- docs: Documentation

Scenario 4: Finding Files and Information

Let's say you want to find all Python files in your project:

$ find ~/projects/myapp -name "*.py"

If you had Python files, it would list them. But you don't have any yet.

Search for text in a file:

$ grep "Flask" README.md

Flask

It finds and shows the line containing Flask.

Search in all files:

$ grep -r "API" ~/projects/myapp

README.md:- backend: Python Flask API

This recursively searches all files in the directory for the word API.

Scenario 5: Working with Permissions

You want a script to run. First create it:

$ echo '#!/bin/bash' > script.sh
$ echo 'echo "Hello from script"' >> script.sh

Try to run it:

$ ./script.sh

bash: ./script.sh: Permission denied

You created it but didn't make it executable. Check permissions:

$ ls -la script.sh

-rw-r--r-- 1 hashimo hashimo 43 Dec 11 17:10 script.sh

The x (execute) permission is missing. Add it:

$ chmod +x script.sh

Now check:

$ ls -la script.sh

-rwxr-xr-x 1 hashimo hashimo 43 Dec 11 17:10 script.sh

Notice the x is now present for owner, group, and everyone. Run it:

$ ./script.sh

Hello from script

It works!

Scenario 6: Managing Disk Space

You wonder how much space your project uses:

```
$ du -sh ~/projects/myapp

4.2M  /home/hashimo/projects/myapp
```

4.2 megabytes. Not bad. Check disk usage system-wide:

```
$ df -h
```

This shows:

```
Filesystem  Size  Used Avail Use% Mounted on
/dev/sda1  50G  20G  30G  40% /
/dev/sda2  10G  9.5G  0.5G  95% /home
```

Your main filesystem has 50GB total, 20GB used, 30GB available. Your home directory is on a separate partition that's 95% full. Time to clean up some old files!

Scenario 7: Installing Application Dependencies

You want to develop with Python. You need pip (Python package manager) and a virtual environment:

```
$ sudo apt install python3-pip python3-venv
```

Then in your backend directory, create a virtual environment:

```
$ cd ~/projects/myapp/backend
$ python3 -m venv env
```

This creates an isolated Python environment for your project. Activate it:

```
$ source env/bin/activate
```

Your prompt changes to:

(env) hashimo@ubuntu:~/projects/myapp/backend$

The (env) tells you you're in a virtual environment. Now install your project dependencies:

$ pip install flask

This installs Flask using pip. Since you're in a virtual environment, it doesn't affect your system Python. If you activate another environment, it has different packages.

When you're done:

$ deactivate

You're back to the system Python.

Scenario 8: Viewing System Information

Check what OS you're using:

$ lsb_release -a

No LSB modules are available.
Distributor ID: Ubuntu
Release: 20.04
Codename: focal

You have Ubuntu 20.04 (Focal Fossa).

Check your CPU:

$ nproc

4

You have 4 CPU cores available.

Check memory:

$ free -h

total used free shared buff/cache available
Mem: 15Gi 5.2Gi 7.8Gi 280Mi 2.0Gi 9.0Gi
Swap: 2.0Gi 0B 2.0Gi

You have 15GB of RAM, 5.2GB is currently in use, 7.8GB is free.

Check your user:

$ whoami

hashimo

Check groups you belong to:

$ groups

hashimo adm cdrom sudo dip plugdev

You're a member of several groups. The sudo group allows you to use sudo without entering a password for certain commands.

SECTION 6: THE SHELL AND SCRIPTING - AUTOMATION AND POWER

Why Shell Scripting Matters

A shell script is a text file containing commands. Instead of typing each command individually, you can write them once, save them to a file, and run them whenever you need. This is how you automate repetitive tasks.

The Power of Pipes

One of Unix's greatest ideas is pipes. A pipe (|) connects the output of one program to the input of another.

For example, imagine you have a file with thousands of lines and you want to find specific ones, count them, and save to a file:

```
$ cat huge_file.txt | grep "error" | wc -l > error_count.txt
```

Breaking this down:

- cat huge_file.txt outputs the file contents
- grep "error" filters lines containing "error"
- wc -l counts the matching lines
- > error_count.txt saves the result to a file

Three programs work together through pipes. Each does one thing well.

Real Example: Processing Log Files

Imagine you have a web server with logs:

/var/log/nginx/access.log

This file has millions of lines, one per request, looking like:

10.0.0.50 - - [11/Dec/2025:17:23:45 +0000] "GET /api/users HTTP/1.1" 200 1234

You want to know which IP addresses are making the most requests:

```
$ cat /var/log/nginx/access.log | cut -d' ' -f1 | sort | uniq -c | sort -rn | head -10
```

This pipeline:

- cat outputs the entire log
- cut -d' ' -f1 extracts the first field (separated by spaces) - the IP
- sort arranges them
- uniq -c counts occurrences of each unique IP
- sort -rn sorts in reverse order (most first)
- head -10 shows the top 10

Output might be:

```
 5432 10.0.0.50
 3421 10.0.0.51
 2234 10.0.0.52
```

You immediately see your most active users.

Writing Your First Script

Create a file called backup.sh:

```
#!/bin/bash

SOURCE=$HOME/projects
DEST=$HOME/backups
DATE=$(date +%Y-%m-%d)

mkdir -p $DEST
```

```
tar -czf $DEST/backup-$DATE.tar.gz $SOURCE

echo "Backup completed: $DEST/backup-$DATE.tar.gz"
```

Let's break this down:

- #!/bin/bash tells the system to run this with bash
- SOURCE=$HOME/projects stores the directory to backup in a variable
- DEST=$HOME/backups stores the destination
- DATE=$(date +%Y-%m-%d) captures today's date (2025-12-11 format)
- mkdir -p $DEST creates the backup directory if it doesn't exist
- tar -czf creates a compressed backup file
- echo shows a confirmation message

Save it, make it executable:

```
$ chmod +x backup.sh
```

Run it:

```
$ ./backup.sh
```

Backup completed: /home/hashimo/backups/backup-2025-12-11.tar.gz

Every time you run it, you get a new dated backup. If you run it tomorrow, you get a separate file for that day.

Real Example: System Maintenance Script

Create a script that cleans up old files:

```
#!/bin/bash
```

```
echo "Starting cleanup..."

# Remove temporary files
rm -rf /tmp/*
echo "Cleaned /tmp"

# Remove old log files
find /var/log -type f -name "*.old" -delete
echo "Cleaned old logs"

# Update package lists
apt update
echo "Updated packages"

# Remove unused packages
apt autoremove -y
echo "Removed unused packages"

echo "Cleanup completed!"
```

This script:

- Removes temporary files from /tmp
- Removes old log files (must be run with sudo)
- Updates package lists
- Automatically removes packages nothing depends on

Run it with sudo:

```
$ sudo ./cleanup.sh

Starting cleanup...
Cleaned /tmp
Cleaned old logs
Updated packages
Reading package lists... Done
Removed unused packages
Cleanup completed!
```

Real Example: Monitoring a Server

Create a script that checks if your web server is running:

```
#!/bin/bash

if systemctl is-active --quiet nginx; then
 echo "Nginx is running ✓"
else
 echo "Nginx is DOWN - restarting..."
 sudo systemctl restart nginx
 echo "Nginx restarted"
fi
```

This script:

- Checks if nginx is running with systemctl is-active
- If it is, shows a success message
- If it's not, restarts it automatically
- Shows what happened

You can run this every minute with a cron job. If your server ever stops, it automatically restarts. You'll know because the script logged what happened.

SECTION 7: SYSTEM ADMINISTRATION - KEEPING UBUNTU RUNNING

Managing Users

Ubuntu is meant for multiple users. Each user has:

- A username

- A password
- A home directory
- Permissions on files

Create a new user:

$ sudo adduser alice

You'll see:

Adding user `alice' ...
Adding new group `alice' (1001) ...
Adding new home directory `/home/alice' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for alice
[Enter] to skip any of these

A new account is created with a home directory. The new user can log in.

Check all users:

$ cat /etc/passwd | head

This shows:

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/bin:/usr/sbin/nologin
sync:x:4:4:sync:/bin:/bin/sync
hashimo:x:1000:1000:Hashimo:/home/hashimo:/bin/bash
alice:x:1001:1001:Alice,,,:/home/alice:/bin/bash

Each line is a user account. Format: username:password:uid:gid:info:home:shell

Let alice run administrative commands by adding her to the sudo group:

$ sudo usermod -aG sudo alice

Now alice can use sudo.

Remove a user:

$ sudo deluser --remove-home alice

This removes alice and her home directory.

Managing Services

Services are programs that run in the background. Your web server is a service. Your database is a service. The system init (systemd) manages them.

See all services:

$ systemctl list-units --type=service

You'll see hundreds of services. Some are running, some aren't.

Check a specific service:

$ systemctl status nginx

You see:

● nginx.service - A high performance web server and a reverse proxy server

Loaded: loaded (/lib/systemd/system/nginx.service; enabled; vendor preset: enabled)
Active: active (running) since Thu 2025-12-11 17:45:23 UTC; 2h 15min ago
Process: 1234 ExecStartPre=/usr/sbin/nginx -t (code=exited, status=0/SUCCESS)
Process: 1235 ExecStart=/usr/sbin/nginx (code=exited, status=0/SUCCESS)
Main PID: 1236 (nginx)

Active: running - it's on.

Stop it:

$ sudo systemctl stop nginx

Start it again:

$ sudo systemctl start nginx

Make it auto-start at boot:

$ sudo systemctl enable nginx

Disable auto-start:

$ sudo systemctl disable nginx

Real Example: Setting Up a Database User

Your web application needs a database user. Connect to PostgreSQL as root:

$ sudo -u postgres psql

You're in the PostgreSQL interactive terminal:

postgres=#

Create a user:

postgres=# CREATE USER webapp WITH PASSWORD 'securepassword';

Create a database:

postgres=# CREATE DATABASE myapp_db OWNER webapp;

Give the user permissions:

postgres=# GRANT ALL PRIVILEGES ON DATABASE myapp_db TO webapp;

Exit:

postgres=# \q

Now your application can connect to the database using the webapp user. The database is only accessible to that user.

Real Example: Monitoring Disk Space

You want to find what's using disk space:

$ du -sh /home/hashimo/*

This shows:

```
2.5G  /home/hashimo/Downloads
3.2G  /home/hashimo/Documents
1.8G  /home/hashimo/projects
0.5G  /home/hashimo/Videos
```

Downloads is taking 2.5GB. Look deeper:

```
$ ls -lah /home/hashimo/Downloads | head -20
```

Find large files:

```
$ find /home/hashimo/Downloads -type f -size +100M
```

This shows files larger than 100MB. Delete ones you don't need:

```
$ rm /home/hashimo/Downloads/old_movie.mp4
```

Real Example: System Updates

Ubuntu sends security patches regularly. Check what needs updating:

```
$ sudo apt update
$ apt list --upgradable
```

Shows:

```
nginx/focal-updates 1.18.0-0ubuntu1.4 amd64 [upgradable from: 1.18.0-0ubuntu1.3]
postgresql/focal-updates 12+208.pgdg20.04+1 amd64 [upgradable from: 12+207.pgdg20.04+1]
```

Update everything:

```
$ sudo apt upgrade
```

Or update only specific packages:

```
$ sudo apt install nginx postgresql
```

The system keeps you current with security fixes and improvements.

SECTION 8: YOUR JOURNEY FROM NOVICE TO LINUX MASTER

What You've Accomplished So Far

You've made it past the hardest part. Seriously. The moment a person understands that Linux isn't magic - that it's just files, permissions, and processes following logical rules - everything else becomes learnable. You've reached that moment.

You can now:

- Navigate the filesystem without getting lost
- Install software without breaking your system
- Create and run scripts
- Understand what's happening when you run a command
- Read error messages and understand what they mean
- Find information when you don't know something

This is massive. This is the difference between being intimidated by the command line and feeling at home in it.

But You're Still At The Beginning

Here's the honest truth: you're now competent at the basics. You're like someone who learned to drive in an empty parking lot. You can go forward, backward, and turn. But you haven't driven on the highway yet.

The next months are where the real learning happens. Not because the concepts are harder - they're actually simpler in many ways - but because you're going to learn to apply them to actual problems. You're going to learn the patterns that real

system administrators use. You're going to develop the intuition that separates people who follow tutorials from people who understand systems.

The Progression That Makes Sense

The path forward isn't random. It follows a logical sequence where each level builds on the previous one. Each level answers a question:

Level 1: How do I work efficiently with text?
Level 2: How do I connect systems together?
Level 3: What's actually running, and how do I manage it?
Level 4: How do I automate complex tasks?
Level 5: When things break, how do I fix them?
Level 6: How do I set up systems properly from the start?
Level 7: How do I protect systems from attacks?
Level 8: How do I make systems run fast?

But Here's The Thing About This Path

Every single level is immediately useful. You're not learning theory for a test you'll take in six months. You're learning things you'll use tomorrow.

Someone finishes Level 1 (text processing) and suddenly they can extract information from log files that would take others an hour. Someone finishes Level 2 (networking) and they can manage servers from their laptop. Someone finishes Level 4 (advanced scripting) and they can automate work that used to take them days.

This is where people get addicted to Linux. Not because Linux

is cool (it is), but because they start to feel genuinely powerful. They realize they're not fighting the system - they're working with it. Problems that seemed impossible become simple.

Level 1: Text Processing - The Superpower

Next comes the most practical, immediately-useful skill: working with text.

Here's why this comes first: almost everything in Linux is text. Configuration files are text. Log files are text. Data is text. Code is text. When you need to find something, you're searching through text. When you need to modify something, you're editing text.

Most people treat text processing as a side skill. Wrong. It's the foundation of productivity.

You already know grep for finding text. Now you'll learn:

sed - the stream editor that transforms text with surgical precision. You'll see a thousand-line log file and extract exactly what you need.

awk - which is almost a programming language disguised as a text processor. With awk, you'll process data the way databases do, but faster for small data.

Regular expressions - the language that describes patterns. Once you learn regex, you stop thinking about "find and replace." You think about "any email address format" or "dates in this pattern" or "anything that looks like a phone number."

Vim - the most powerful text editor ever created. Yes, it's weird. Yes, it takes time to learn. And yes, once you know it, you'll be 10x faster than people using graphical editors.

Real application? A developer can take a million-line log file and extract exactly the error messages they need, count them by type, and save the results in 30 seconds using a pipeline. Someone without these skills would need half an hour and a spreadsheet application.

This level teaches you that Linux isn't slow - it's incredibly fast when you know how to work with it.

Level 2: Networking - Your Systems Connect

After mastering text, everything becomes network-oriented.

Here's reality: nobody has a single server anymore. Everything connects. Your laptop connects to the cloud. Your services connect to databases in other countries. Your phone connects to multiple networks simultaneously.

At this level you learn:

How networks actually work - IP addresses, DNS, ports. Not networking certification depth. The depth of "I understand why my application can't reach the database."

SSH - the tool you'll use thousands of times. Secure Shell is how you access any remote system. It's encrypted, it's standard, and it's been unchanged for 25 years because it's that good. You'll learn how SSH keys work, why they're more secure than passwords, and how to set them up so you never need to type a password again.

Network troubleshooting - ping, traceroute, netstat, ss. These tools answer the question: can I reach that server? If not, where does the connection fail?

Firewall basics - how to open ports so services can communicate, and how to close ports so attackers can't get in.

Why this matters: after this level, you'll be able to access any server you have permission to access, troubleshoot connection problems, understand why services can or can't communicate. You'll realize that the network is just another system following logical rules.

Most importantly, you'll understand SSH well enough that you could manage 100 servers from your laptop securely.

Level 3: Process Management - The System Under Control

Now you understand how to communicate across networks. Next you need to understand what's happening on those systems.

This level teaches you how to see what's actually running:

ps and related tools show you every process on the system
top and htop give you real-time views of what's using resources
Signals let you communicate with processes - pause them, resume them, shut them down cleanly
Resource monitoring reveals where your bottlenecks are

This is when things click. You'll be running a command, the system will slow to a crawl, and instead of rebooting in frustration, you'll:

1. Open top
2. See that PostgreSQL is using 95% CPU
3. Realize there's a runaway query

4. Kill it
5. Continue working

You went from being a victim of system behavior to being in control of it.

You also learn about background jobs, how to run multiple things simultaneously, and how to manage them. This is when you realize bash isn't just a shell - it's a complete job management system.

Level 4: Advanced Scripting - Automation Is Mastery

After three levels, you understand the fundamentals, networks, and processes. Now comes the power: automation.

You learn:

Loops that let you run commands hundreds or thousands of times
Conditionals that make decisions based on system state
Functions that let you reuse code
Error handling that makes your scripts robust enough for production
Cron jobs that run scripts on a schedule

This is where people have their "aha" moments. You write a 20-line script that does work you used to spend an hour on manually. You schedule it to run every night. Suddenly you have 5 hours back every week.

You write a script that monitors your application. If it ever goes down, the script detects it and restarts it automatically. Your application never experiences downtime again.

You write a backup script that runs every night, and keeps the

last 30 days of backups, automatically deleting old ones. Your data is safe and you never think about it again.

This level is where the word "impossible" disappears from your vocabulary. If there's a repetitive task, you automate it. That's not being lazy - that's being productive.

Level 5: Troubleshooting - When Things Break

You've been learning for weeks. Now comes the level that teaches you what to do when reality breaks your expectations.

Because here's the thing: everything breaks eventually. Services crash. Disk fills up. Connections time out. Permissions get messed up. You need a systematic way to figure out what's wrong.

You learn:

Where to look for log files (and there are conventions - you're not searching blind)
How to read log files (they usually say exactly what's wrong)
Systematic troubleshooting methodology (figure out what changed, narrow down the problem, test hypotheses)
Common problems and their solutions (you'll recognize patterns)
How to ask for help effectively (when you get stuck, showing the right information gets you answers)

This is the level where you stop being afraid of errors. Errors are just information. Your system is trying to tell you what's wrong. You've just learned to listen.

Level 6: Configuration Management - Building Right From The Start

After months of learning, you've learned to fix problems. Now learn to prevent them.

You understand:

How systemd unit files work (the files that tell the system how to start services)

Configuration file conventions (where settings go, how they're formatted)

How to properly configure applications (the right way vs the "it works" way)

Best practices for system setup

This is when you go from "systems I can manage" to "systems that manage themselves." You set up a service properly, it runs flawlessly for years. You misconfigure it, and you're hunting for issues forever.

The difference between good system administrators and great ones? The great ones set things up right the first time so they don't have to fix them later.

Level 7: Security - Nothing Matters If You're Compromised

You've been learning for months. Now add the critical dimension: security.

You learn:

User and group management (not just creating users, but understanding roles and permissions)
File permissions in detail (the octal notation, special bits, setuid, setgid)
SSH key management (how to create keys, protect them, use

them)
Sudo and privilege escalation (doing sensitive operations safely)
Firewall configuration (basic rules that protect your system)
Security best practices (the patterns that keep attackers out)

This is when you realize that every previous lesson has security implications. Running a script as root? That's dangerous. Storing a password in a config file? That's dangerous. Running a service as root when it only needs read permissions? That's dangerous.

Security isn't something you add at the end. It's something you're thinking about from the beginning.

Level 8: Performance and Optimization - Making Things Fast

Last comes optimization. And it comes last for a reason.

Most people try to optimize things that aren't broken. That's wasted effort. After mastering Levels 1-7, you understand:

How to identify actual bottlenecks (not guessing)
How to measure improvements (so you know if your change helped)
The trade-offs between different optimizations
How systems scale as load increases

You learn about:

CPU optimization
Memory optimization
Disk I/O optimization
Network optimization
Kernel tuning parameters

But more importantly, you learn when NOT to optimize. "It's fast enough" is a valid conclusion. You don't optimize what isn't a problem.

This level is where you go from "my system works" to "my system works and handles 100 times more load than it did six months ago."

The Timeline - And Why It Matters

Honestly? This progression takes about 5-6 months of consistent effort. Real effort. Not "I watched a YouTube video" effort. Actually doing the work.

But here's what you'll be at the end:

You won't need tutorials anymore. You'll understand systems. You'll read documentation and know exactly what it means. You'll encounter a problem you've never seen before and solve it because you understand the fundamentals.

You'll be able to manage dozens of servers. You'll be able to build infrastructure. You'll be able to diagnose and fix problems that have other people completely stumped.

That's not learning how to use Linux. That's becoming someone who understands operating systems.

And honestly? The journey is the reward. Every level teaches you something that makes you more powerful. Every skill you learn is something you'll use literally every day. You're not learning for a certification test - you're learning things that will make you better at your job (or career, or hobby) immediately.

LEVEL 1: TEXT PROCESSING - THE SUPERPOWER THAT CHANGES EVERYTHING

There's a moment in every Linux user's journey where they stop fighting the system and start dancing with it.

It happens when you realize you can do in seconds what takes someone with a graphical tool an hour.

You have a 500MB log file. You need to find every error from the last day, group them by error type, and count how many times each appears. A normal person opens it in a spreadsheet tool. It crashes. They try again. It hangs. They give up.

You type one command:

```
grep "$(date +%Y-%m-%d)" error.log | awk '{print $5}' | sort | uniq -c | sort -rn | head -10
```

Thirty seconds later, you have the answer.

That's when you realize the command line isn't just faster. It's a different way of thinking. You stop being a person using a computer and start being someone who controls it.

This is Level 1. This is where that power comes from.

GREP: FINDING PATTERNS EVERYWHERE

grep is the most fundamental text search tool. It finds lines matching a pattern.

Basic search:

```
grep "error" application.log
```

Breaking down:

grep - Global Regular Expression Print (search tool)
"error" - Pattern to search for (literal string)
application.log - File to search in

Outputs every line containing "error".

Search with line numbers:

```
grep -n "error" application.log
```

Breaking down:

grep - Search tool
-n - Number flag. -n = show line numbers. Tells grep to print line number before each match
"error" - Pattern
application.log - File

Output shows:

```
42:error: connection timeout
157:error: database unreachable
203:error: authentication failed
```

The numbers on the left are line numbers.

Count matches instead of showing them:

```
grep -c "error" application.log
```

Breaking down:

grep - Search tool
-c - Count flag. -c = count matches. Shows total count instead of lines
"error" - Pattern
application.log - File

Output:

```
234
```

Exactly 234 lines match.

Search backwards (invert match):

```
grep -v "debug" application.log
```

Breaking down:

grep - Search tool
-v - Verbose invert. -v = invert match. Show lines that DON'T match
"debug" - Pattern to exclude
application.log - File

Shows every line that doesn't contain "debug".

Case-insensitive search:

```
grep -i "ERROR" application.log
```

Breaking down:

grep - Search tool
-i - Insensitive flag. -i = case-insensitive. Matches "ERROR", "Error", "error", etc.
"ERROR" - Pattern
application.log - File

Search in multiple files:

```
grep "error" /var/log/*.log
```

Breaking down:

grep - Search tool
"error" - Pattern
/var/log/*.log - Multiple files (* = wildcard)

Search recursively (subdirectories):

```
grep -r "TODO" ~/projects
```

Breaking down:

grep - Search tool
-r - Recursive flag. -r = search in subdirectories. Searches all files in ~/projects and subdirectories
"TODO" - Pattern
~/projects - Directory to search

Show lines before and after match:

```
grep -A 5 -B 5 "error" application.log
```

Breaking down:

grep - Search tool
-A 5 - After flag. -A = show lines after match. 5 = number of lines

-B 5 - Before flag. -B = show lines before match. 5 = number of lines
"error" - Pattern
application.log - File

Shows matched line plus 5 lines before and 5 lines after (context).

Combine options:

```
grep -rni "warning" /var/log
```

Breaking down:

grep - Search tool
-r - Recursive (search subdirectories)
-n - Number (show line numbers)
-i - Insensitive (case-insensitive)
"warning" - Pattern
/var/log - Directory

Search with regular expressions:

```
grep -E "error|warning|critical" application.log
```

Breaking down:

grep - Search tool
-E - Extended regex. -E = use extended regular expressions. Allows | (or), +, ?, (), etc.
"error|warning|critical" - Pattern (error OR warning OR critical)
application.log - File

Modern shortcut:

```
egrep "error|warning|critical" application.log
```

Breaking down:

egrep - Extended grep (shorthand for grep -E)
"error|warning|critical" - Extended regex pattern
application.log - File

Only matching part of line:

grep -o "ERROR[0-9]*" application.log

Breaking down:

grep - Search tool
-o - Only matching flag. -o = output only the matching part, not the whole line
"ERROR[0-9]*" - Pattern (ERROR followed by zero or more digits)
application.log - File

Output shows only the matches:

ERROR123
ERROR42
ERROR999

SED: TRANSFORMING TEXT

sed is a stream editor. It transforms text based on commands.

Simple substitution:

sed 's/old/new/' file.txt

Breaking down:

sed - Stream editor
's/old/new/' - Substitution command
 s = substitute (replace)
 / = delimiter (separates parts)
 old = pattern to find
 new = replacement
 / = end delimiter
file.txt - File to edit

This replaces FIRST occurrence of "old" with "new" on each line. Does NOT modify the file, just shows output.

Replace ALL occurrences on each line:

sed 's/old/new/g' file.txt

Breaking down:

sed - Stream editor
's/old/new/g' - Substitution command
 s = substitute
 /old/new/ = pattern and replacement
 g = global flag (replace ALL on line, not just first)
file.txt - File

Global flag is critical. Without it, only first occurrence per line is replaced.

Edit file in place:

sed -i 's/old/new/g' file.txt

Breaking down:

sed - Stream editor

-i - In-place flag. -i = modify file directly (don't output to console)
's/old/new/g' - Replace all old with new
file.txt - File to modify

Backup before modifying:

```
sed -i.bak 's/old/new/g' file.txt
```

Breaking down:

sed - Stream editor
-i.bak - In-place with backup. -i = modify file. .bak = backup extension
's/old/new/g' - Substitution command
file.txt - File

Creates file.txt.bak (original) and modifies file.txt.

Use different delimiter (when pattern contains slashes):

```
sed 's|/path/old|/path/new|g' file.txt
```

Breaking down:

sed - Stream editor
's|/path/old|/path/new|g' - Substitution with pipe delimiter
 s = substitute
 | = delimiter (instead of /)
 /path/old = pattern (contains slashes, so using | as delimiter)
 /path/new = replacement
 g = global
file.txt - File

Why use different delimiter? Because the pattern contains

slashes. Using slash as delimiter would confuse sed.

Delete lines matching pattern:

```
sed '/pattern/d' file.txt
```

Breaking down:

sed - Stream editor
'/pattern/d' - Address and command
 /pattern/ = lines matching this pattern
 d = delete (remove these lines)
file.txt - File

Shows all lines EXCEPT those matching pattern.

Only show specific lines:

```
sed -n '10,20p' file.txt
```

Breaking down:

sed - Stream editor
-n - Quiet flag. -n = don't print automatically. Only print lines specified by p command
'10,20p' - Address and print command
 10,20 = lines 10 through 20
 p = print (show these lines)
file.txt - File

Shows only lines 10-20.

Show specific lines and delete others:

```
sed -n '5p;10p;15p' file.txt
```

Breaking down:

sed - Stream editor
-n - Quiet (don't print automatically)
'5p;10p;15p' - Multiple commands separated by semicolons
 5p = print line 5
 10p = print line 10
 15p = print line 15
file.txt - File

Shows only lines 5, 10, and 15.

AWK: PROCESSING STRUCTURED DATA

awk processes data with fields and records. Think of it as a database tool for text.

Basic awk (print all lines):

awk '{print}' file.txt

Breaking down:

awk - Text processing language
'{print}' - Program (in single quotes)
 {print} = print every line
file.txt - Input file

Equivalent to cat file.txt.

Print specific field:

awk '{print $1}' file.txt

Breaking down:

awk - Text processing language
'{print $1}' - Program
$1 = first field
file.txt - File

Fields are separated by whitespace by default.

Input:

```
john 25 engineer
mary 30 manager
bob 22 intern
```

Output:

```
john
mary
bob
```

Print multiple fields:

```
awk '{print $1, $3}' file.txt
```

Breaking down:

awk - Text processing language
'{print $1, $3}' - Program
$1, $3 = first and third fields
, = comma separator in output
file.txt - File

Output:

```
john engineer
mary manager
bob intern
```

Specify field separator:

```
awk -F: '{print $1}' /etc/passwd
```

Breaking down:

awk - Text processing language
-F: - Field separator flag. -F = field separator. : = use colon as separator
'{print $1}' - Program (first field)
/etc/passwd - File (uses colons to separate fields)

/etc/passwd has format: username:password:uid:gid:...

Print with custom separator:

```
awk -F: '{print $1 " is user " $3}' /etc/passwd
```

Breaking down:

awk - Text processing language
-F: - Use colon as field separator
'{print $1 " is user " $3}' - Program (concatenate text and fields)
/etc/passwd - File

Output:

```
root is user 0
daemon is user 1
bin is user 2
```

Filter lines by condition:

```
awk '$2 > 25 {print $1, $2}' employees.txt
```

Breaking down:

awk - Text processing language
'$2 > 25' - Condition (where second field is greater than 25)
'{print $1, $2}' - Action (print first and second fields)
employees.txt - File

Shows only employees older than 25.

Count occurrences:

```
awk '{count[$2]++} END {for (item in count) print item,
count[item]}' file.txt
```

Breaking down:

awk - Text processing language
'{count[$2]++}' - For each line, increment counter for field 2
$2 = the field to count by
++ = increment
count[] = array (associative array)
'END {for (item in count) print item, count[item]}' - After all lines, print results
END = execute after all input
for (item in count) = loop through array
print item, count[item] = print the item and its count
file.txt - File

Sum values by category:

```
awk '{sum[$1] += $2} END {for (item in sum) print item,
sum[item]}' data.txt
```

Breaking down:

awk - Text processing language
'{sum[$1] += $2}' - For each line, add field 2 to total for

field 1
sum[$1] += $2 = add field 2 to sum array for field 1
'END {for (item in sum) print item, sum[item]}' - Print results
data.txt - File

REGULAR EXPRESSIONS: THE PATTERN LANGUAGE

Regular expressions (regex) are patterns that match text flexibly.

Meta-characters:

. = any single character
* = zero or more of previous
+ = one or more of previous
? = zero or one of previous
^ = start of line
$ = end of line
[abc] = any of a, b, or c
[a-z] = any letter
[0-9] = any digit
[^abc] = anything EXCEPT a, b, c

Find lines starting with error:

grep "^error" application.log

Breaking down:

grep - Search tool
"^error" - Pattern
^ = start of line
error = literal text
application.log - File

Find lines ending with a number:

```
grep "[0-9]$" application.log
```

Breaking down:

grep - Search tool
"[0-9]$" - Pattern
[0-9] = digit
$ = end of line
application.log - File

Find email addresses:

```
grep -E "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" file.txt
```

Breaking down:

grep - Search tool
-E - Extended regex (allows +, {}, etc.)
"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}" - Pattern
[a-zA-Z0-9._%+-]+ = one or more valid email characters
@ = literal @
[a-zA-Z0-9.-]+ = domain name
\. = literal period (escaped)
[a-zA-Z]{2,} = 2 or more letters (TLD)
file.txt - File

Find IP addresses:

```
grep -E "[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" access.log
```

Breaking down:

grep - Search tool
-E - Extended regex
"[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}" - Pattern

[0-9]{1,3} = 1 to 3 digits
\. = literal period
(repeat 4 times for IP address)
access.log - File

VIM: THE EDITOR THAT TRANSFORMS YOUR SPEED

vim is a powerful text editor. The learning curve is steep, but the payoff is massive.

Two modes:

1. Insert mode - Type text (like a normal editor)
2. Command mode - Execute commands (search, replace, delete, etc.)

Open file in vim:

vim filename.txt

Breaking down:

vim - Vi Improved text editor
filename.txt - File to edit

You're in Command mode. Press i to enter Insert mode.

In Insert mode:

Type normally. Escape to return to Command mode.

In Command mode (don't type colons, just the letter):

i - Insert (enter Insert mode before cursor)
a - Append (insert after cursor)
o - Open new line (below current)
dd - Delete line
yy - Yank (copy line)
p - Paste
u - Undo
: - Colon (start typing commands)

Search and replace in vim:

::%s/old/new/g

Breaking down:

: - Start command
% - All lines (% = entire file)
s/old/new/g - Substitute command
 s = substitute
 /old/new/ = pattern and replacement
 g = global (all on line)

Save and quit:

:wq

Breaking down:

: - Command mode
w = write (save)
q = quit

Quit without saving:

:q!

Breaking down:

: - Command
q = quit
! = force (ignore unsaved changes)

REAL-WORLD TEXT PROCESSING WORKFLOWS

Workflow 1: Analyzing Web Server Logs

You have an nginx access log. You want top 10 endpoints.

```
cat /var/log/nginx/access.log | awk '{print $7}' | sort | uniq -c |
sort -rn | head -10
```

Breaking down the pipeline:

cat /var/log/nginx/access.log - Read log file
| - Pipe (send to next)
awk '{print $7}' - Extract field 7 (the endpoint path)
$7 = 7th field in nginx log
| sort - Sort all paths
| uniq -c - Count unique paths. -c = count. Shows how many times each appears
| sort -rn - Re-sort numerically descending. -r = reverse. -n = numeric
| head -10 - Show first 10

Output:

```
 1450 /api/users
 980 /static/images
 654 /api/products
```

...

Workflow 2: Finding Slow Database Queries

You have a query log. Find queries taking over 1 second.

```
grep -E "Query_time: [1-9]|Query_time: [0-9]{2,}" mysql.log | head
-20
```

Breaking down:

grep - Search tool
-E - Extended regex
"Query_time: [1-9]|Query_time: [0-9]{2,}" - Pattern (1-9 seconds OR 2+ digit seconds)
mysql.log - Log file
| head -20 - Show first 20

Workflow 3: Extract IPs from Log File

Get all unique IPs and their request counts.

```
awk '{print $1}' /var/log/nginx/access.log | sort | uniq -c | sort -rn |
head -20
```

Breaking down:

awk '{print $1}' - Extract first field (IP address)
/var/log/nginx/access.log - Log file
| sort - Sort IPs
| uniq -c - Count occurrences of each IP
| sort -rn - Sort by count (highest first)
| head -20 - Show top 20

Workflow 4: Clean Configuration File

Remove comments and empty lines from config:

```
grep -v "^#" /etc/nginx/nginx.conf | grep -v "^$" | cat -n
```

Breaking down:

```
grep -v "^#" - Remove lines starting with #
 -v = invert (show what doesn't match)
 ^# = lines starting with #
/etc/nginx/nginx.conf - Config file
| grep -v "^$" - Remove empty lines
 ^$ = start and end with nothing (empty line)
| cat -n - Show with line numbers
```

Workflow 5: Count Errors by Type

Group errors by type, count frequency.

```
grep "error" application.log | awk -F: '{print $2}' | sort | uniq -c |
sort -rn
```

Breaking down:

```
grep "error" - Find error lines
application.log - Log file
| awk -F: '{print $2}' - Extract field 2 (using : as separator)
| sort - Sort
| uniq -c - Count
| sort -rn - Sort by count
```

Workflow 6: Extract Specific Date Range

Show logs from specific date.

```
awk 'BEGIN{skip=1} /2024-01-15/{skip=0} /2024-01-16/
{skip=1} skip{next} {print}' access.log
```

Breaking down:

awk 'BEGIN{skip=1}' - Start with skip=1 (skip lines)
BEGIN = execute before processing
'/2024-01-15/{skip=0}' - When you find this date, set skip=0 (show lines)
'/2024-01-16/{skip=1}' - When you find next date, set skip=1 (skip again)
'skip{next}' - If skip is 1, go to next line
'{print}' - Otherwise print the line
access.log - File

COMBINING TOOLS: PIPES AND POWER

The real power comes from piping tools together.

tool1 | tool2 | tool3

Breaking down:

| - Pipe operator. Sends output of one tool to input of next
tool1 - First tool processes input
| - Send to next
tool2 - Second tool processes tool1's output
| - Send to next
tool3 - Final tool shows result

Example: Complex log analysis

cat access.log | grep "2024-01" | awk '{print $1}' | sort | uniq -c | sort -rn | grep -E "[0-9]{3,}" | wc -l

Breaking down each part:

cat access.log - Read log
| grep "2024-01" - Filter for January 2024
| awk '{print $1}' - Extract IP addresses
| sort - Sort IPs
| uniq -c - Count each IP
| sort -rn - Sort by count (highest first)
| grep -E "[0-9]{3,}" - Filter IPs with 100+ requests (3+ digit numbers)
| wc -l - Count how many IPs have 100+ requests

This finds how many IPs made 100+ requests in January.

COMMON TEXT PROCESSING MISTAKES

Mistake 1: Forgetting to Escape Special Characters

You write:

```
sed 's/price = $100/price = $50/' file.txt
```

The $ has special meaning (end of line). Use different delimiter:

```
sed 's|price = $100|price = $50|' file.txt
```

Breaking down:

sed - Stream editor
's|...|...|' - Use pipe as delimiter (not slash)
file.txt - File

Or escape the special character:

```
sed 's/price = \$100/price = \$50/' file.txt
```

Breaking down:

\ = escape character (tells sed treat next char literally)
$100 = now treated as literal text

Mistake 2: Using sed Without Testing

You write:

```
sed -i 's/old/new/g' important.txt
```

If the regex is wrong, you corrupt your file. Always test first:

```
sed 's/old/new/g' important.txt | head
```

Breaking down:

(same sed command)
| head - Show first few lines of output
(verify before using -i flag)

Mistake 3: Forgetting Global Flag in sed

You write:

```
sed 's/a/b/' file.txt
```

This replaces only FIRST occurrence per line. Usually you want:

```
sed 's/a/b/g' file.txt
```

Breaking down:

g = global flag (all occurrences per line)

Mistake 4: awk Field Numbering Issues

Your data is inconsistent:

```
name : 30 : engineer
alice 40 programmer
```

Without checking field separators first, awk breaks.

Check before writing awk:

```
head -5 file.txt
```

See the format. Then write appropriate awk:

```
awk -F: '{print $1}' file.txt (if colon-separated)
awk '{print $1}' file.txt (if space-separated)
```

Mistake 5: grep Without -E for Complex Patterns

You write:

```
grep "error|warning" file.txt
```

Returns nothing. | doesn't work without -E:

```
grep -E "error|warning" file.txt
```

Breaking down:

-E = extended regex (enables |, +, ?, etc.)

LEVEL 1 CHECKPOINT: YOU HAVE TRANSFORMED YOUR CAPABILITY

What You Can Do Now

You can search any file for patterns
You can transform text at scale
You can process structured data like a database
You can combine tools into powerful pipelines
You can understand every flag in every command
You can read documentation and know what it means
You can analyze logs in seconds
You can automate text manipulation
You can explain why you use each flag

The Transformation

Text processing is where most software work happens. Databases are text. Configuration files are text. Logs are text. Your ability to work with text at speed is your foundation.

You've gone from "I need a tool for this" to "I can combine these tools."

What Comes Next

You now have power over text on a single system. Next comes connecting systems, understanding networks, and working across multiple machines.

LEVEL 2: NETWORKING - YOUR SYSTEMS CONNECT

Here's reality: nobody has a single server anymore.

You might have a laptop at home, a server in the cloud, a database on another server, and a web frontend somewhere else. They all need to talk. They all need to be fast. They all need to be secure.

And if something breaks in the network, your entire stack collapses silently.

This is where Level 2 changes everything.

In Level 1, you learned to manipulate text on a single system. You were king of that one machine.

Now you're going to learn to see the connections between systems. You're going to understand why a connection succeeds or fails. You're going to fix network problems in minutes instead of hours of searching.

Let's start with the fundamentals.

UNDERSTANDING YOUR NETWORK INTERFACE - EVERY FLAG EXPLAINED

See your network interfaces:

ip link show

Breaking down:

ip - Network configuration tool (modern, replaces ifconfig)
link - Link-layer (hardware) information
show - Display them (default action)

See IP addresses:

ip addr show

Breaking down:

ip - Network tool
addr - Address subsystem (IP addresses, not hardware)
show - Display

Show just IPv4:

ip -4 addr show

Breaking down:

ip - Network tool
-4 - Filter flag. -4 = IPv4 only (excludes IPv6)
addr show - Address subsystem, display

Show single interface:

ip addr show enp0s3

Breaking down:

ip - Network tool
addr show - Show addresses
enp0s3 - Interface name (argument, not a flag)

Set an IP address:

sudo ip addr add 10.0.0.50/24 dev enp0s3

Breaking down:

sudo - Run as root (needed for network config changes)
ip - Network tool
addr - Address subsystem
add - Action (add new address)
10.0.0.50/24 - IP address and netmask (CIDR notation)
dev enp0s3 - Device flag. dev = which device. enp0s3 = interface

Bring interface up:

sudo ip link set enp0s3 up

Breaking down:

sudo - Root privilege
ip - Network tool
link - Link layer
set - Modify
enp0s3 - Interface
up - Action (activate)

Bring interface down:

sudo ip link set enp0s3 down

Breaking down:

sudo ip link set enp0s3 - Same structure
down - Deactivate (offline)

DNS CONFIGURATION AND QUERIES

See DNS servers:

cat /etc/resolv.conf

Breaking down:

cat - Display file contents
/etc/resolv.conf - Resolver configuration file

Test DNS resolution:

nslookup example.com

Breaking down:

nslookup - Name server lookup tool
example.com - Domain to look up

Modern tool with more detail:

dig example.com

Breaking down:

dig - Domain Information Groper (modern DNS tool)
example.com - Domain query

Shorter output:

dig +short example.com

Breaking down:

dig - DNS tool

+short - Output flag. + = enable flag. short = brief output
example.com - Domain

Reverse DNS (IP to hostname):

dig -x REMOTE_SERVER_IP

Breaking down:

dig - DNS tool
-x - Reverse lookup mode (IP to name instead of name to IP)
REMOTE_SERVER_IP - IP address to look up

Query specific record type:

dig example.com MX

Breaking down:

dig - DNS tool
example.com - Domain
MX - Record type (mail exchange)

Record types: A (address), AAAA (IPv6), CNAME (alias), NS (nameserver), TXT (text)

SSH: SECURE REMOTE ACCESS WITH EVERY FLAG

Basic connection:

ssh user@server.example.com

Breaking down:

ssh - Secure Shell client
user - Remote username
@ - Separator
server.example.com - Hostname/IP

Generate SSH key:

```
ssh-keygen -t ed25519 -b 4096 -f ~/.ssh/id_ed25519
```

Breaking down:

ssh-keygen - Generate SSH key pair
-t ed25519 - Type flag. -t = encryption type. ed25519 = modern encryption
-b 4096 - Bits flag. -b = strength. 4096 = key size (bits)
-f ~/.ssh/id_ed25519 - File flag. -f = save location. ~/ = home. id_ed25519 = filename

Copy public key to server:

```
ssh-copy-id -i ~/.ssh/id_ed25519.pub user@server.example.com
```

Breaking down:

ssh-copy-id - Copy SSH public key to remote
-i ~/.ssh/id_ed25519.pub - Identity flag. -i = which public key file
user@server.example.com - Remote server

Connect with specific key:

```
ssh -i ~/.ssh/id_ed25519 user@server.example.com
```

Breaking down:

ssh - Secure Shell
-i ~/.ssh/id_ed25519 - Identity flag. -i = which private key
user@server.example.com - Server

Non-standard port:

```
ssh -p 2222 user@server.example.com
```

Breaking down:

ssh - Secure Shell
-p 2222 - Port flag. -p = which port. 2222 = port number
user@server.example.com - Server

Verbose output (debugging):

```
ssh -v user@server.example.com
```

Breaking down:

ssh - Secure Shell
-v - Verbose flag. Shows connection details
user@server.example.com - Server

Add -vv or -vvv for more detail.

Run command on remote:

```
ssh user@server.example.com "whoami"
```

Breaking down:

ssh - Secure Shell
user@server.example.com - Server
"whoami" - Command (must be quoted)

Port forwarding (SSH tunnel):

ssh -L 3306:127.0.0.1:3306 user@remote.example.com

Breaking down:

ssh - Secure Shell
-L - Local forward flag. -L = forward local port to remote
3306 - Local port
: - Separator
127.0.0.1 - Remote destination (from remote's perspective)
3306 - Remote port
user@remote.example.com - Tunnel through this server

PING AND TRACEROUTE - DIAGNOSTIC TOOLS

Test connectivity:

ping example.com

Breaking down:

ping - Send ICMP packets (test reachability)
example.com - Target

Ping count (stop after N):

ping -c 5 example.com

Breaking down:

ping - Ping tool
-c 5 - Count flag. -c = number of packets. 5 = stop after 5
example.com - Target

Ping interval:

ping -c 5 -i 2 example.com

Breaking down:

ping - Ping tool
-c 5 - Send 5 packets
-i 2 - Interval flag. -i = wait between packets. 2 = 2 seconds
example.com - Target

Trace route:

traceroute example.com

Breaking down:

traceroute - Show path to destination
example.com - Target

Using ICMP instead of UDP:

traceroute -I example.com

Breaking down:

traceroute - Trace tool
-I - Use ICMP protocol (capital I)
example.com - Target

SOCKET STATISTICS - SEEING NETWORK CONNECTIONS

Show listening sockets:

ss -tulpn

Breaking down EACH FLAG:

ss - Socket Statistics tool
-t - TCP protocol
-u - UDP protocol
-l - Listening sockets only
-p - Program flag. Show which process owns connection
-n - Numeric flag. Don't resolve hostnames (faster)

Show all connections:

ss -tupn

Breaking down:

ss - Socket Statistics
-t - TCP
-u - UDP
-p - Program
-n - Numeric

No -l, so shows listening AND established.

Show summary:

ss -s

Breaking down:

ss - Socket Statistics
-s - Summary flag. Overall statistics

Find what's using a port:

ss -tulpn | grep :4000

Breaking down:

ss -tulpn - All listening sockets
| - Pipe to next tool
grep :4000 - Search for port 4000

FIREWALL - UFW WITH COMPLETE FLAG EXPLANATIONS

Enable firewall:

sudo ufw enable

Breaking down:

sudo - Run as root
ufw - Uncomplicated Firewall
enable - Turn it on

Check status:

sudo ufw status

Breaking down:

sudo ufw - Firewall tool
status - Show current rules

Status with numbers:

sudo ufw status numbered

Breaking down:

sudo ufw status - Show rules
numbered - Add rule numbers (for deleting)

Allow port:

sudo ufw allow 22

Breaking down:

sudo ufw - Firewall
allow - Allow incoming
22 - Port number

Allow from IP:

sudo ufw allow from EXAMPLE_IP

Breaking down:

sudo ufw - Firewall
allow - Allow traffic
from EXAMPLE_IP - Only from this IP

Deny traffic:

sudo ufw deny 23

Breaking down:

sudo ufw - Firewall
deny - Reject
23 - Port

Default policies:

sudo ufw default deny incoming

sudo ufw default allow outgoing

Breaking down:

sudo ufw default - Set default rule
deny incoming - Reject by default
allow outgoing - Accept outgoing by default

Delete rule:

sudo ufw delete allow 22

Breaking down:

sudo ufw - Firewall
delete - Remove rule
allow 22 - Which rule (as it was added)

SYSTEMCTL - SERVICE MANAGEMENT WITH FLAGS

Check service status:

sudo systemctl status nginx

Breaking down:

sudo - Root privilege
systemctl - System Control (service manager)
status - Action (show status)
nginx - Service name

Start service:

sudo systemctl start nginx

Breaking down:

sudo systemctl - Service tool
start - Action (start)
nginx - Service

Stop service:

sudo systemctl stop nginx

Breaking down:

sudo systemctl - Service tool
stop - Action (stop)
nginx - Service

Restart:

sudo systemctl restart nginx

Breaking down:

sudo systemctl - Service tool
restart - Stop then start
nginx - Service

Reload (apply config changes without stopping):

sudo systemctl reload nginx

Breaking down:

sudo systemctl - Service tool
reload - Graceful reload (keep running)
nginx - Service

Enable on boot:

sudo systemctl enable nginx

Breaking down:

sudo systemctl - Service tool
enable - Auto-start at boot
nginx - Service

Disable on boot:

sudo systemctl disable nginx

Breaking down:

sudo systemctl - Service tool
disable - Don't auto-start
nginx - Service

Check if enabled:

sudo systemctl is-enabled nginx

Breaking down:

sudo systemctl - Service tool
is-enabled - Query auto-start status
nginx - Service

View service logs:

sudo journalctl -u nginx

Breaking down:

sudo journalctl - View system logs

-u nginx - Unit flag. -u = service name. nginx = which service
(shows all logs for that service)

Last 20 lines:

sudo journalctl -u nginx -n 20

Breaking down:

sudo journalctl -u nginx - Service logs
-n 20 - Number flag. -n = last N lines. 20 = show last 20

Follow in real-time:

sudo journalctl -u nginx -f

Breaking down:

sudo journalctl -u nginx - Service logs
-f - Follow flag. Show new lines as they appear (like tail -f)

SSH KEY SETUP FOR AUTOMATION

Generate automation key (no passphrase):

ssh-keygen -t ed25519 -f ~/.ssh/deploy_key -N ""

Breaking down:

ssh-keygen - Generate key
-t ed25519 - Type
-f ~/.ssh/deploy_key - File location
-N "" - No passphrase. -N = passphrase flag. "" = empty string

Copy to servers:

```
for server in server1.example.com server2.example.com; do
 ssh-copy-id -i ~/.ssh/deploy_key.pub user@$server
done
```

Breaking down:

for server in ... do - Loop through servers
ssh-copy-id - Copy public key
-i ~/.ssh/deploy_key.pub - Identity flag
user@$server - Remote server
done - End loop

Use in automation:

```
ssh -i ~/.ssh/deploy_key user@server.example.com "commands
here"
```

Breaking down:

ssh - Secure Shell
-i ~/.ssh/deploy_key - Which private key
user@server.example.com - Server
"commands here" - Command to run

NETWORK TROUBLESHOOTING WORKFLOW

Problem: Website is slow from one location

Step 1 - Test connectivity:

```
ping -c 5 example.com
```

Breaking down:

ping - Test reachability
-c 5 - Send 5 packets
example.com - Target

Step 2 - Check DNS:

nslookup example.com

Breaking down:

nslookup - Lookup hostname
example.com - Domain

Step 3 - Trace path:

traceroute example.com

Breaking down:

traceroute - Show route
example.com - Destination

Step 4 - Check local services:

ss -tulpn

Breaking down:

ss - Socket stats
-tulpn - All listening TCP/UDP with program names

KEY PRINCIPLES FOR LEVEL 2

1. Every flag has meaning. Understand each one.

2. Flags go BEFORE arguments:

ssh -p 2222 -i ~/.ssh/key user@server (CORRECT)
ssh user@server -p 2222 -i ~/.ssh/key (WRONG)

3. Man pages are your friend:

man ssh - Read the manual
ssh --help - Quick reference

4. Test before production:

ping first to verify connectivity
traceroute to understand the path
ss to see what's listening

5. Comments in scripts:

When you create scripts, comment why you use each flag:

ssh -v user@server # -v for debugging if connection slow

LEVEL 3: PROCESS MANAGEMENT - WHAT'S RUNNING AND WHY

You now understand text on one machine (Level 1) and

connecting machines (Level 2).

But you still don't see what's happening RIGHT NOW on your systems.

Right now, your computer is running dozens or hundreds of processes. Some are critical. Some are wasting CPU and memory. Some are stuck and broken.

You don't know any of this.

This is where Level 3 changes things.

You're going to learn to see what's actually running. You'll understand CPU usage, memory, processes, signals, and resource management.

This is where you realize your system is not a mystery. It's transparent. You can see everything.

LISTING PROCESSES - WHAT'S RUNNING RIGHT NOW

See all running processes:

```
ps aux
```

Breaking down:

```
ps - Process Status (show running processes)
aux - Standard flags combined
 a = all users (not just your processes)
 u = user format (show owner, memory, CPU usage)
 x = no controlling terminal (show background processes)
```

Output shows:

```
USER  PID %CPU %MEM  VSZ  RSS TTY STAT START  TIME COMMAND
root  1  0.0  0.2  10644  5456 ?  Ss  Dec11  0:05 /sbin/init
root  456  0.0  0.1  9876  3456 ?  Ss  Dec11  0:02 /lib/systemd/systemd-journald
user  1234  2.5  1.2 234567 56789 ?  Sl  10:30  0:45 /usr/bin/firefox
```

Breaking it down:

USER = who is running it
PID = process ID (unique number)
%CPU = percentage of CPU it's using
%MEM = percentage of system memory it's using
VSZ = virtual memory size
RSS = resident set size (actual RAM used)
TTY = controlling terminal (? means background)
STAT = state (Ss = sleeping/system, Sl = sleeping/has threads, R = running)
START = when it started
TIME = total CPU time used
COMMAND = what it is

Filter for specific process:

```
ps aux | grep firefox
```

Breaking down:

ps aux - List all processes
| - Pipe to next command
grep firefox - Search for "firefox"

Modern alternative using pgrep:

pgrep -f firefox

Breaking down:

pgrep - Process grep (search for processes)
-f - Full command flag. -f = search full command line (not just process name)
firefox - What to search for

Shows just the PID:

1234

TOP AND HTOP - REALTIME MONITORING

Real-time process monitoring:

top

Breaking down:

top - Table of Processes (live, interactive monitoring)

Shows continuously updating list sorted by CPU usage.

Press 'q' to quit.

Press 'm' to sort by memory.

Press 'p' to sort by PID.

htop is better (more visual):

htop

Breaking down:

htop - Human Top (improved version of top)

More features:

- Color coding
- Easier navigation
- Better memory display
- Tree view of processes
- Interactive killing of processes

Sort by column:

htop -s PERCENT_CPU

Breaking down:

htop - Process monitor
-s PERCENT_CPU - Sort flag. -s = sort by. PERCENT_CPU = CPU usage

UNDERSTANDING PROCESS STATE

Process states in ps output:

D = Uninterruptible sleep (waiting for I/O)
R = Running or runnable
S = Sleeping (interruptible)
T = Stopped or traced
Z = Zombie (exited but not cleaned up)

Zombie processes:

A zombie is a process that has exited but its parent hasn't cleaned it up.

Check for zombies:

ps aux | grep defunct

Breaking down:

ps aux - All processes
| grep defunct - Search for "defunct" (zombie indicator)

Kill zombie parent:

ps aux | grep PPID

Breaking down:

ps aux - Process listing
| grep PPID - Find parent process ID of zombie
(then kill the parent, which cleans up zombie)

SIGNALS AND PROCESS CONTROL

Signals are ways to communicate with running processes.

Kill a process (SIGTERM - graceful):

kill 1234

Breaking down:

kill - Send signal to process
1234 - PID to send signal to

Default is SIGTERM (signal 15) - allows process to clean up.

Send specific signal:

kill -15 1234

Breaking down:

kill - Send signal
-15 - Signal number. -15 = SIGTERM (terminate gracefully)
1234 - PID

Common signals:

1 = SIGHUP (hangup, reload config)
2 = SIGINT (interrupt, like Ctrl+C)
9 = SIGKILL (kill immediately, can't be caught)
15 = SIGTERM (terminate, graceful)
19 = SIGSTOP (stop/pause)
18 = SIGCONT (continue after stop)

Send SIGKILL (force kill):

kill -9 1234

Breaking down:

kill - Send signal
-9 - Signal number. -9 = SIGKILL (cannot be ignored)
1234 - PID to kill

Kill by name:

pkill firefox

Breaking down:

pkill - Process kill (kill by name, not PID)
firefox - Process name to kill

Kill with pattern:

pkill -f "python.*script"

Breaking down:

pkill - Kill by process
-f - Full command flag
"python.*script" - Pattern to match

Kill all instances:

pkill -9 -f "old_service"

Breaking down:

pkill - Kill by name
-9 - Signal 9 (SIGKILL)
-f - Full command matching
"old_service" - Pattern

BACKGROUND AND FOREGROUND JOBS

Run command in background:

```
command &
```

Breaking down:

command - Whatever command to run
& - Background operator (returns prompt immediately)

Example:

```
python3 long_script.py &
```

Breaking down:

python3 long_script.py - Command to run
& - Run in background

Output:

```
[1] 5678
```

[1] = job number
5678 = PID

List background jobs:

```
jobs
```

Breaking down:

jobs - Show background jobs for current shell

Output:

```
[1]+  Running  python3 long_script.py
[2]-  Stopped  vim file.txt
```

+ = current job

- = previous job

Bring job to foreground:

fg %1

Breaking down:

fg - Foreground (bring to front)
%1 - Job number flag

Stop foreground job:

Press Ctrl + Z

This sends SIGSTOP (pause, not kill).

Resume in background:

bg %1

Breaking down:

bg - Background (resume in background)
%1 - Job number

Resume in foreground:

fg %1

Breaking down:

fg - Foreground (bring to front)
%1 - Job number

MEMORY AND CPU ANALYSIS

Show memory usage:

free -h

Breaking down:

free - Show memory statistics
-h - Human readable. -h = display in GB/MB (not bytes)

Output:

total used free shared buff/cache available
Mem: 15Gi 7.2Gi 2.1Gi 456Mi 5.6Gi 7.1Gi
Swap: 2.0Gi 256Mi 1.7Gi

Show system uptime and load:

uptime

Breaking down:

uptime - System uptime and load average

Output:

10:45:32 up 45 days, 3:21, 2 users, load average: 0.45, 0.52, 0.48

Load average explanation:

0.45 = average over last 1 minute
0.52 = average over last 5 minutes
0.48 = average over last 15 minutes

On a 4-core system, load 1.0 = one core fully used.

Show CPU info:

lscpu

Breaking down:

lscpu - List CPU information

or

cat /proc/cpuinfo

Breaking down:

cat - Display file contents
/proc/cpuinfo - Virtual file with CPU details

MONITORING SPECIFIC PROCESSES

Watch process with watch command:

watch -n 1 'ps aux | grep firefox'

Breaking down:

watch - Repeatedly run command with updates
-n 1 - Number flag. -n = interval. 1 = every 1 second
'ps aux | grep firefox' - Command to repeat

Updates every second showing latest process info.

Monitor file descriptor usage:

lsof -p 1234

Breaking down:

lsof - List open files
-p 1234 - Process flag. -p = which PID. 1234 = process to check

Shows all files, sockets, pipes opened by that process.

Monitor memory of specific process:

top -p 1234

Breaking down:

top - Process monitor
-p 1234 - PID flag. -p = which process. 1234 = only this PID

Real-time monitoring of single process.

PROCESS DEPENDENCIES AND PARENT-CHILD RELATIONSHIPS

Show process tree:

pstree

Breaking down:

pstree - Process tree (show parent-child relationships)

Output:

```
systemd
□□□ systemd-journald
□□□ systemd-udevd
□□□ nginx
□  □□□ nginx (worker)
□  □□□ nginx (worker)
□□□ sshd
```

Show tree with PIDs:

```
pstree -p
```

Breaking down:

pstree - Process tree
-p - PID flag. -p = show process IDs

Show full details:

```
pstree -ap
```

Breaking down:

pstree - Process tree
-a - Show arguments (command line arguments)
-p - Show PIDs

Find parent of process:

```
ps -o ppid= -p 1234
```

Breaking down:

ps - Process status
-o ppid= - Output flag. -o = output format. ppid = parent process ID. = = suppress header
-p 1234 - PID flag

Shows just the parent PID of process 1234.

PROCESS RESOURCE LIMITS

Show limits for shell:

ulimit -a

Breaking down:

ulimit - User limits (show or set resource limits)
-a - All flag. -a = show all limits

Output:

core file size 0
data seg size unlimited
scheduling priority 0
file size unlimited
pending signals 31829
file descriptors 1024
memory limit unlimited
stack size 8388608
cpu time unlimited
max user processes 31829
virtual memory unlimited

Set limit (file size):

ulimit -f 1000000

Breaking down:

ulimit - User limits
-f 1000000 - File size flag. -f = max file size. 1000000 = blocks

Set memory limit:

ulimit -v 1000000

Breaking down:

ulimit - User limits
-v 1000000 - Virtual memory flag. -v = max memory. 1000000 = kilobytes

NICE AND PRIORITY

Run process with low priority:

nice -n 19 command

Breaking down:

nice - Set priority (run with adjusted niceness)
-n 19 - Nice flag. -n = niceness level. 19 = lowest priority
command - Command to run

Priority ranges from -20 (highest) to 19 (lowest).

Check priority of running process:

ps -o pid,cmd,nice -p 1234

Breaking down:

ps - Process status
-o pid,cmd,nice - Output flag. -o = format. Shows PID, command, nice value
-p 1234 - PID flag

Change priority of running process:

renice -n 10 -p 1234

Breaking down:

renice - Re-nice (change running process priority)
-n 10 - Nice flag. -n = new niceness. 10 = lower priority
-p 1234 - PID flag

REAL-WORLD PROCESS MANAGEMENT WORKFLOWS

Workflow 1: Find Resource Hog

Problem: System is slow. What's using resources?

top -b -n 1 | head -15

Breaking down:

top - Process monitor

-b - Batch flag. -b = batch mode (output once, don't update)
-n 1 - Number flag. -n = number of iterations. 1 = run once
| head -15 - Show first 15 lines

Or using htop:

```
htop -o PERCENT_CPU
```

Breaking down:

htop - Interactive monitor
-o PERCENT_CPU - Sort flag. -o = order by. PERCENT_CPU = CPU usage

Workflow 2: Stop Hung Process Gracefully

Process is stuck. Try graceful kill first:

```
kill -15 PID
sleep 5
if ps -p PID > /dev/null; then
 kill -9 PID
fi
```

Breaking down:

kill -15 PID - Send SIGTERM (graceful)
sleep 5 - Wait 5 seconds
if ps -p PID - Check if still running
 kill -9 PID - Force kill if still there

Workflow 3: Monitor Long-Running Job

Start job and monitor:

```
python3 long_job.py &
sleep 2
```

```
watch -n 2 'ps aux | grep long_job'
```

Breaking down:

python3 long_job.py & - Start in background
sleep 2 - Wait for it to start
watch -n 2 - Monitor every 2 seconds
'ps aux | grep long_job' - Show process info

Workflow 4: Find Process Using Port

Problem: Port 4000 is in use. What's using it?

```
lsof -i :4000
```

Breaking down:

lsof - List open files
-i :4000 - Internet flag. -i = network. :4000 = which port

Or using ss:

```
ss -tlnp | grep 4000
```

Breaking down:

ss - Socket statistics
-tlnp - TCP, listening, numeric, program names
| grep 4000 - Filter for port 4000

Workflow 5: Monitor Memory Leak

Application using more memory over time:

```
watch -n 10 'ps aux | grep app | grep -v grep'
```

Breaking down:

watch -n 10 - Update every 10 seconds
'ps aux | grep app | grep -v grep' - Show app process

Watch RSS column (actual memory).

COMMON PROCESS MANAGEMENT MISTAKES

Mistake 1: Using kill -9 First

You write:

kill -9 stubborn_process

This forces immediate death without cleanup. The process can't save state, release locks, or clean up resources.

What you should do:

kill stubborn_process (SIGTERM first)
sleep 5
kill -9 stubborn_process (only if still running)

Breaking down:

kill - Send SIGTERM (graceful)
(wait for cleanup)
kill -9 - Only force if necessary

Mistake 2: Killing Wrong Process

You want to kill firefox but grep for "firefox":

pkill firefox

This might catch other things (firefox-esr, firefox-bin, etc.).

Be specific:

```
pkill -f "^/usr/bin/firefox"
```

Breaking down:

pkill - Kill by name
-f - Full command matching
"^/usr/bin/firefox" - Exact path (^ = start of line)

Mistake 3: Ignoring Zombie Processes

You see zombies in ps output but ignore them.

Zombies don't use much memory, but they waste PIDs. Fix them:

Find parent:

```
ps aux | grep '<defunct>'
```

Kill parent:

```
kill -9 parent_pid
```

Breaking down:

kill -9 - Force kill
parent_pid - Parent of zombie processes

Mistake 4: Running CPU-Intensive Task at Normal Priority

Your backup script runs at normal priority and slows system:

```
nice -n 19 /path/to/backup.sh
```

Breaking down:

nice - Low priority
-n 19 - Niceness 19 (very low priority)
/path/to/backup.sh - Run this script

Now system stays responsive.

Mistake 5: Not Monitoring Resource Usage

System gets slow. You don't know why until it's critical.

Set up monitoring:

```
watch -n 5 'top -b -n 1 | head -10'
```

Breaking down:

watch -n 5 - Check every 5 seconds
'top -b -n 1 | head -10' - Show top 10 processes

LEVEL 4: ADVANCED SCRIPTING - AUTOMATING COMPLEXITY

You now understand text (Level 1), networks (Level 2), and what's running (Level 3).

But you're still typing commands one at a time.

This is where you stop being a user and become an engineer.

Level 4 teaches you to automate. To write scripts that do what took you hours in minutes. To create systems that run without you.

This is where the real power comes from.

BASH SCRIPT STRUCTURE

Every script starts the same way:

```
#!/bin/bash

set -euo pipefail
```

Breaking down:

#!/bin/bash - Shebang (which interpreter to use)
#! = special marker (execute following)
/bin/bash = path to bash interpreter
set -euo pipefail - Safety flags
set - Shell options
-e - Exit on error (stop if any command fails)
-u - Exit on undefined variable (stop if var doesn't exist)
-o pipefail - Exit on pipe error (if any command in pipeline fails)

This makes scripts fail safely instead of continuing with errors.

Basic script structure:

```
#!/bin/bash
set -euo pipefail

# Variables
BACKUP_DIR="/backups"
SOURCE_DIR="/data"

# Functions
backup_database() {
 echo "Backing up database..."
 # code here
}

# Main
main() {
 backup_database
}

main "$@"
```

Breaking down:

#!/bin/bash - Interpreter
set -euo pipefail - Safety settings
Comments - Lines starting with # are ignored
VARIABLE=value - Variable assignment (no spaces around =)
function_name() { } - Function definition
main "$@" - Call main function with arguments ($@ = all arguments passed to script)

VARIABLES AND PARAMETER EXPANSION

Assign variable:

```
NAME="John"
AGE=30
TIMESTAMP=$(date +%s)
```

Breaking down:

NAME="John" - Variable assignment (string)
AGE=30 - Variable assignment (number)
TIMESTAMP=$(date +%s) - Command substitution. $() = run command, use output

Use variable:

```
echo "Hello $NAME, you are $AGE years old"
```

Breaking down:

$NAME - Variable expansion (replaced with value)
$AGE - Variable expansion

Output:

Hello John, you are 30 years old

Parameter expansion (advanced):

```
${NAME:-default}
```

Breaking down:

${NAME:-default} - If NAME not set, use "default"

Length of variable:

```
${#NAME}
```

Breaking down:

${#NAME} - Length of NAME (number of characters)

Substring:

${NAME:0:4}

Breaking down:

${NAME:0:4} - Extract substring
0 = start position
4 = length

Substitute in variable:

${NAME//o/0}

Breaking down:

${NAME//o/0} - Replace all o with 0
// = replace all (single / = replace first only)

CONDITIONALS - IF/ELSE

Simple if statement:

```
if [ -f /etc/passwd ]; then
 echo "File exists"
fi
```

Breaking down:

if - Start conditional

[-f /etc/passwd] - Test condition (-f = file exists test)
then - Execute if true
echo "File exists" - Command to run
fi - End conditional

File tests:

-f = file exists
-d = directory exists
-s = file exists and not empty
-r = file readable
-w = file writable
-x = file executable

If-else:

```
if [ "$NAME" = "John" ]; then
 echo "Hello John"
else
 echo "Hello stranger"
fi
```

Breaking down:

if - Start
["$NAME" = "John"] - String comparison (= checks equality)
then - If true
echo "Hello John" - Execute
else - Otherwise
echo "Hello stranger" - Execute
fi - End

String comparisons:

= - Equal
!= - Not equal
-z - Empty (zero length)

-n - Not empty

Numeric comparisons:

-eq - Equal
-ne - Not equal
-lt - Less than
-le - Less than or equal
-gt - Greater than
-ge - Greater than or equal

If numeric:

```
if [ "$AGE" -gt 18 ]; then
 echo "Adult"
fi
```

Breaking down:

["$AGE" -gt 18] - Numeric greater than test
-gt = greater than (for numbers)

Multiple conditions:

```
if [ "$AGE" -gt 18 ] && [ "$NAME" = "John" ]; then
 echo "Adult John"
fi
```

Breaking down:

&& - AND operator (both must be true)
(could also use single [... -a ...] but && is clearer)

Multiple conditions OR:

```
if [ "$COLOR" = "red" ] || [ "$COLOR" = "blue" ]; then
 echo "Primary color"
```

```
fi
```

Breaking down:

|| - OR operator (either must be true)

Negation:

```
if ! [ -f /etc/passwd ]; then
 echo "File does not exist"
fi
```

Breaking down:

! - NOT operator (negates condition)

LOOPS - FOR AND WHILE

For loop over list:

```
for file in /tmp/*.log; do
 echo "Processing $file"
 rm "$file"
done
```

Breaking down:

for - Loop construct
file - Variable (loop variable, created automatically)
in /tmp/*.log - List to loop over (glob expands to files)
do - Start loop body
echo "Processing $file" - Loop body commands
done - End loop

For loop over range:

```
for i in {1..10}; do
 echo "Number $i"
done
```

Breaking down:

for i in {1..10} - Loop from 1 to 10
do - Start
echo "Number $i" - Print number
done - End

For loop over array:

```
FILES=("file1.txt" "file2.txt" "file3.txt")
for file in "${FILES[@]}"; do
 echo "$file"
done
```

Breaking down:

FILES=(...) - Array assignment (parentheses for array)

for file in "${FILES[@]}" - Loop over array
"${FILES[@]}" = all elements of FILES array
[@] = all elements
do - Loop body
done - End

While loop:

```
count=1
while [ "$count" -le 10 ]; do
 echo "Count: $count"
 ((count++))
done
```

Breaking down:

while - Loop while condition is true
["$count" -le 10] - While count <= 10
do - Loop body
((count++)) - Increment count
((...)) = arithmetic expansion
++ = increment operator
done - End

Break from loop:

```
for i in {1..100}; do
 if [ "$i" = "50" ]; then
 break
 fi
 echo "$i"
done
```

Breaking down:

break - Exit loop immediately

Continue to next iteration:

```
for i in {1..10}; do
 if [ "$i" = "5" ]; then
 continue
 fi
 echo "$i"
done
```

Breaking down:

continue - Skip to next iteration (skip 5)

FUNCTIONS - REUSABLE CODE

Define function:

```
greet() {
 echo "Hello $1"
}

greet "John"
```

Breaking down:

greet() { } - Function definition
$1 = first argument passed to function
greet "John" - Call function with argument

Output:

```
Hello John
```

Function with multiple arguments:

```
add() {
 echo $(($1 + $2))
}

add 5 3
```

Breaking down:

add() - Define function
$1 - First argument
$2 - Second argument
$((...)) - Arithmetic expansion

add 5 3 - Call with two arguments

Output:

8

Function with return value:

```
is_even() {
 if [ $(($1 % 2)) -eq 0 ]; then
 return 0  # Success (true)
 else
 return 1  # Failure (false)
 fi
}

if is_even 4; then
 echo "Even"
fi
```

Breaking down:

return 0 - Success (true) exit code
return 1 - Failure (false) exit code
if is_even 4 - Use function as condition (0 = true)

Function with local variables:

```
increment() {
 local counter=$1
 ((counter++))
 echo "$counter"
}
```

Breaking down:

local - Variable scope flag. -local makes variable local to

function
$1 - First argument
((counter++)) - Increment

Function returning data:

```
get_timestamp() {
 echo "$(date +%s)"
}

TIMESTAMP=$(get_timestamp)
```

Breaking down:

get_timestamp() - Function that outputs data
echo "..." - Function returns via echo
TIMESTAMP=$(get_timestamp) - Capture output in variable

ERROR HANDLING - TRAP AND CLEANUP

Handle errors with trap:

```
#!/bin/bash
set -euo pipefail

cleanup() {
 echo "Cleaning up..."
 rm -f /tmp/temp_file
}

trap cleanup EXIT

echo "Script running"
```

Breaking down:

cleanup() { } - Cleanup function
trap cleanup EXIT - Execute cleanup on exit
 trap = set trap handler
 cleanup = function to call
 EXIT = signal/event (execute on script exit)

Trap on error:

```
#!/bin/bash
set -euo pipefail

error_handler() {
 echo "Error on line $1"
 exit 1
}

trap 'error_handler $LINENO' ERR
```

Breaking down:

error_handler - Function called on error
$LINENO - Line number where error occurred
trap 'error_handler $LINENO' ERR - Call on error
 ERR = error signal (command fails)

Exit with error code:

```
if [ ! -f /etc/passwd ]; then
 echo "Password file not found"
 exit 1
fi
```

Breaking down:

exit 1 - Exit with code 1 (error)
exit 0 - Exit with code 0 (success)

Check command exit code:

```
if ! command; then
 echo "Command failed with code $?"
fi
```

Breaking down:

$? = exit code of last command
! = negate (true if command failed)

ARRAYS AND DATA STRUCTURES

Create array:

```
SERVERS=("web1" "web2" "web3")
```

Breaking down:

SERVERS=(...) - Array assignment (parentheses)

Access array:

```
echo "${SERVERS[0]}"
```

Breaking down:

${SERVERS[0]} - First element (index 0)
[@] would mean all elements

Loop over array:

```
for server in "${SERVERS[@]}"; do
 echo "Server: $server"
done
```

Breaking down:

"${SERVERS[@]}" - All array elements
each iteration sets $server to one element

Add to array:

```
SERVERS+=("web4")
```

Breaking down:

+= - Append to array

Array length:

```
${#SERVERS[@]}
```

Breaking down:

${#SERVERS[@]} - Number of elements in array

Associative array (like dictionary):

```
declare -A PORTS
PORTS[web]=80
PORTS[https]=443
PORTS[ssh]=22
```

Breaking down:

declare -A - Declare associative array

-A = associative (key-value)

PORTS[web]=80 - Set key-value

Loop over associative array:

```
for service in "${!PORTS[@]}"; do
 echo "$service: ${PORTS[$service]}"
done
```

Breaking down:

```
${!PORTS[@]} - All keys in array
${PORTS[$service]} - Get value for key
```

REAL-WORLD SCRIPT PATTERNS

Pattern 1: Backup Script with Timestamp

```
#!/bin/bash
set -euo pipefail

BACKUP_DIR="/backups"
SOURCE="/home/user/data"
TIMESTAMP=$(date +%Y%m%d_%H%M%S)
BACKUP_FILE="$BACKUP_DIR/backup_$TIMESTAMP.tar.gz"

mkdir -p "$BACKUP_DIR"
tar -czf "$BACKUP_FILE" "$SOURCE"
echo "Backup complete: $BACKUP_FILE"
```

Breaking down:

TIMESTAMP=$(date +%Y%m%d_%H%M%S) -

Get current timestamp
BACKUP_FILE="$BACKUP_DIR/backup_$TIMESTAMP.tar.gz" - Construct filename with timestamp

mkdir -p "$BACKUP_DIR" - Create directory if not exists
tar -czf "$BACKUP_FILE" "$SOURCE" - Create compressed archive
 -c = create archive
 -z = gzip compression
 -f = file (specify filename)

Pattern 2: Conditional File Processing

```
#!/bin/bash
set -euo pipefail

for file in /var/log/*.log; do
 if [ -s "$file" ]; then  # -s = file not empty
 size=$(stat -f%z "$file")
 if [ "$size" -gt 1000000 ]; then  # Larger than 1MB
 echo "Rotating $file"
 gzip "$file"
 fi
 fi
done
```

Breaking down:

[-s "$file"] - File exists and not empty
stat -f%z "$file" - Get file size
["$size" -gt 1000000] - If larger than 1MB
gzip "$file" - Compress the file

Pattern 3: Retry Logic

```
#!/bin/bash
set -euo pipefail
```

```
retry() {
 local max_attempts=3
 local attempt=1
 until [ "$attempt" -gt "$max_attempts" ]; do
 if "$@"; then
 return 0
 fi
 echo "Attempt $attempt failed, retrying..."
 ((attempt++))
 sleep 2
 done
 echo "All attempts failed"
 return 1
}

retry curl https://example.com
```

Breaking down:

until - Loop until condition true (opposite of while)
["$attempt" -gt "$max_attempts"] - Until attempt > max
"$@" - Run the command passed to retry
sleep 2 - Wait 2 seconds between retries

Pattern 4: Logging in Scripts

```
#!/bin/bash
set -euo pipefail

LOG_FILE="/var/log/myscript.log"

log() {
 echo "[$(date +'%Y-%m-%d %H:%M:%S')] $1" | tee -a
"$LOG_FILE"
}

log "Script started"
```

```
do_something
log "Script completed"
```

Breaking down:

```
date +'%Y-%m-%d %H:%M:%S' - Format current time
tee -a "$LOG_FILE" - Write to file AND stdout
 -a = append (don't overwrite)
```

Pattern 5: Parsing Command Line Arguments

```
#!/bin/bash
set -euo pipefail

usage() {
 echo "Usage: $0 -f FILE -d DIRECTORY"
 exit 1
}

FILE=""
DIRECTORY=""

while getopts "f:d:" opt; do
 case $opt in
 f) FILE="$OPTARG" ;;
 d) DIRECTORY="$OPTARG" ;;
 *) usage ;;
 esac
done

if [ -z "$FILE" ] || [ -z "$DIRECTORY" ]; then
 usage
fi

echo "File: $FILE, Directory: $DIRECTORY"
```

Breaking down:

getopts - Parse command line options
f:d: = options f and d (colon = they take arguments)
$opt = which option
$OPTARG = argument to option
-z = test if empty

SCHEDULING WITH CRON

Edit crontab:

crontab -e

Breaking down:

crontab - Cron table (schedule tasks)
-e = edit (open in editor)

Cron format:

* * * * * /path/to/command

Breaking down:

* = minute (0-59)
* = hour (0-23)
* = day of month (1-31)
* = month (1-12)
* = day of week (0-7, 0=Sunday)

Examples:

0 2 * * * /path/to/backup.sh

Run backup at 2:00 AM daily

30 */4 * * * /path/to/check.sh

Run check every 4 hours at :30 minutes

0 0 1 * * /path/to/monthly.sh

Run on 1st day of month at midnight

0 9 * * 1-5 /path/to/weekday.sh

Run Monday-Friday at 9 AM

*/5 * * * * /path/to/every5min.sh

Run every 5 minutes

List crontab:

crontab -l

Breaking down:

crontab -l - List current crontab jobs

Remove crontab:

crontab -r

Breaking down:

crontab -r - Remove all crontab entries (dangerous!)

COMMON SCRIPTING MISTAKES

Mistake 1: Not Quoting Variables

You write:

echo $FILENAME

Breaking down:

$FILENAME - Unquoted variable

If FILENAME has spaces, it breaks:

echo "$FILENAME"

Breaking down:

"$FILENAME" - Quoted variable (handles spaces)

Always quote: echo "$var" not echo $var

Mistake 2: Using = in Tests

You write:

if ["$age" -gt 18]; then

Should be:

if ["$age" -gt 18]; then

Breaking down:

-gt = numeric greater than (not =, which is string comparison)

= compares strings. -gt compares numbers.

Mistake 3: Not Setting -e Flag

Without set -e, script continues even if commands fail:

```
set -euo pipefail
```

Breaking down:

-e = exit on error (stop if any command fails)
-u = exit on undefined variable
-o pipefail = fail on pipe error

Always start scripts with this.

Mistake 4: Hardcoding Paths

You write:

```
BACKUP_DIR="/home/user/backups"
```

What if user is different? Use variables:

```
BACKUP_DIR="${HOME}/backups"
```

Breaking down:

${HOME} - Home directory of current user
Much more portable

Mistake 5: No Error Messages

Script fails silently. Use error handling:

```
#!/bin/bash
set -euo pipefail

log_error() {
```

```
 echo "ERROR: $1" >&2
 exit 1
}

if [ ! -f /etc/important ]; then
 log_error "Important file not found"
fi
```

Breaking down:

```
>&2 = redirect error to stderr (not stdout)
exit 1 = exit with error code
```

LEVEL 5: TROUBLESHOOTING - WHEN THINGS BREAK

You've learned to manipulate text, connect systems, manage processes, and automate tasks.

But now everything is breaking.

Services fail. Disk fills up. Networks drop. Processes hang. Permissions are wrong.

This is where Level 5 changes you from a builder to a firefighter.

Troubleshooting is not about knowing answers. It's about knowing how to find answers.

SYSTEMATIC TROUBLESHOOTING APPROACH

The Framework:

1. Observe - What's actually happening?
2. Isolate - What changed? What's different?
3. Diagnose - What's the root cause?
4. Test - Does my theory work?
5. Fix - Implement the solution
6. Verify - Did it actually work?

Do NOT guess or apply random fixes.

CHECKING SYSTEM HEALTH

Full system check (start here):

uptime

Breaking down:

uptime - Load average and system uptime

Shows load and how long system has been running.

Check disk space:

df -h

Breaking down:

df - Disk free (show disk usage)
-h = human readable. -h = show GB/MB (not bytes)

Check specific filesystem:

df -h /

Breaking down:

df -h / - Show usage of root filesystem

Find large files:

du -sh /*

Breaking down:

du - Disk usage
-s = summarize (single total, not per file)
-h = human readable
/* = all in root directory

Check memory:

free -h

Breaking down:

free - Show memory statistics
-h = human readable

Check process status:

ps aux | head

Breaking down:

ps aux - All processes
| head - Show first 10

Check logs:

tail -20 /var/log/syslog

Breaking down:

tail - Show end of file
-20 = last 20 lines
/var/log/syslog - System log file

READING LOGS - FINDING THE PROBLEM

Show recent logs:

journalctl -xe

Breaking down:

journalctl - Journal control (show system logs)
-x = explain (show log entries with explanations)
-e = end (jump to end of logs)

Show logs for service:

journalctl -u nginx

Breaking down:

journalctl - Log viewer
-u nginx = unit (which service). nginx = service name

Show logs with time:

```
journalctl --since "10 minutes ago" --until now
```

Breaking down:

journalctl - Log viewer
--since "10 minutes ago" - Start time for logs
--until now - End time

Show logs from date:

```
journalctl --since "2024-01-15 10:00:00"
```

Breaking down:

journalctl - Log viewer
--since - Start from this datetime

Follow logs in real-time:

```
journalctl -u nginx -f
```

Breaking down:

journalctl -u nginx - Nginx service logs
-f = follow (show new entries as they appear)

Show errors only:

```
journalctl -p err
```

Breaking down:

journalctl - Log viewer
-p err = priority. -p = severity. err = error level

Priority levels:

emerg = system unusable
alert = must act immediately
crit = critical
err = error
warn = warning
notice = normal but significant
info = informational
debug = debug info

Show specific number of entries:

journalctl -u nginx -n 50

Breaking down:

journalctl -u nginx - Service logs
-n 50 = number. -n = last N entries. 50 = show 50

SERVICE TROUBLESHOOTING

Check service status:

sudo systemctl status nginx

Breaking down:

sudo systemctl - Service manager
status nginx - Show status of nginx

If stopped, check why:

sudo journalctl -u nginx -n 20

Breaking down:

journalctl -u nginx - Nginx logs
-n 20 = last 20 entries

Try starting service:

sudo systemctl start nginx

Breaking down:

sudo systemctl - Service manager
start nginx - Start service

If start fails, check errors:

sudo systemctl status nginx

Breaking down:

systemctl status - Shows error message

Check if port is open:

ss -tlnp | grep :80

Breaking down:

ss -tlnp - Listening sockets
| grep :80 - Search for port 80

If port in use by different process:

lsof -i :80

Breaking down:

lsof - List open files
-i :80 = network flag. -i = network. :80 = port

Shows what's using the port.

Restart service:

sudo systemctl restart nginx

Breaking down:

sudo systemctl - Service manager
restart nginx - Stop and start

Check if autostart enabled:

sudo systemctl is-enabled nginx

Breaking down:

systemctl is-enabled - Check autostart status
nginx - Service name

PERMISSION PROBLEMS

Get file permissions:

ls -la /var/www

Breaking down:

ls - List files
-l = long format (shows permissions)
-a = all (show hidden files)

Permission output:

-rw-r--r-- 1 root root 1024 Dec 11 10:30 index.html

Breaking down:

-rw-r--r-- = permissions
- = regular file (d = directory)
rw- = owner (read, write, no execute)
r-- = group (read only)
r-- = others (read only)
1 = link count
root = owner
root = group
1024 = file size

Fix permissions:

chmod 644 /var/www/index.html

Breaking down:

chmod - Change mode (permissions)
644 = permissions (rw-r--r--)
/var/www/index.html - File

Octal notation:

4 = read
2 = write
1 = execute

644 = rw-r--r-- (6 for owner, 4 for group, 4 for others)
755 = rwxr-xr-x (7 for owner, 5 for group, 5 for others)

Make executable:

chmod +x script.sh

Breaking down:

chmod - Change mode
+x = add execute permission
script.sh - File

Change owner:

sudo chown user:group /var/www/index.html

Breaking down:

sudo chown - Change owner
user:group = new owner:group
/var/www/index.html - File

NETWORK TROUBLESHOOTING

Can't reach host:

ping example.com

Breaking down:

ping - Test connectivity
example.com - Target

If no response:

traceroute example.com

Breaking down:

traceroute - Show path to destination
example.com - Target

DNS not resolving:

nslookup example.com

Breaking down:

nslookup - Name server lookup
example.com - Domain

If nslookup fails:

cat /etc/resolv.conf

Breaking down:

cat /etc/resolv.conf - Show DNS config

Check route:

ip route show

Breaking down:

ip route - IP routing command
show - Display routes

Connection refused:

ssh -v user@server.com

Breaking down:

ssh - Secure Shell
-v = verbose (show what's happening)
user@server.com - Server

Slow connection:

mtr example.com

Breaking down:

mtr - My TraceRoute (continuous traceroute)
example.com - Target

Shows packet loss on each hop.

Port unreachable:

nc -zv example.com 80

Breaking down:

nc - Netcat (test connections)
-z = zero I/O (just test connection)
-v = verbose (show result)
example.com - Host
80 - Port

PERFORMANCE TROUBLESHOOTING

System is slow:

top -b -n 1 | head -15

Breaking down:

top - Process monitor
-b = batch mode (output once)
-n 1 = number of iterations. 1 = run once
| head -15 - Show first 15 lines

Shows CPU usage by process.

Check load average:

uptime

Breaking down:

uptime - Shows load average

If load > CPU count, system is overloaded.

Check disk I/O:

iostat -x 1 5

Breaking down:

iostat - IO statistics
-x = extended report
1 = interval (1 second)
5 = number of samples

Shows disk activity.

Check network:

iftop

Breaking down:

iftop - Interface top (bandwidth usage)

Shows which hosts using bandwidth.

Memory leak detection:

ps aux | grep process_name | grep -v grep

Breaking down:

ps aux - All processes
| grep process_name - Filter for process
| grep -v grep - Remove grep itself

Check RSS column for memory growth over time.

REAL-WORLD TROUBLESHOOTING SCENARIOS

Scenario 1: Database Connection Error

Problem: Application says "Cannot connect to database"

Step 1 - Check if database is running:

sudo systemctl status postgresql

Breaking down:

sudo systemctl status - Check service
postgresql - PostgreSQL service

Step 2 - Check port:

```
ss -tlnp | grep 5432
```

Breaking down:

ss -tlnp - Listening ports
| grep 5432 - Database port

Step 3 - Connect locally:

```
psql -U postgres
```

Breaking down:

psql - PostgreSQL client
-U postgres - User. -U = username

Step 4 - Check logs:

```
journalctl -u postgresql -n 20
```

Breaking down:

journalctl -u postgresql - Service logs
-n 20 = last 20 entries

Step 5 - Check network binding:

```
sudo netstat -tlnp | grep 5432
```

Breaking down:

netstat - Network statistics
-tlnp - TCP, listening, numeric, program
| grep 5432 - Database port

If localhost only, that's the problem.

Scenario 2: Disk Full Error

Problem: "No space left on device"

Step 1 - Check disk:

```
df -h
```

Breaking down:

df - Disk free
-h = human readable

Shows which filesystem is full.

Step 2 - Find large files:

```
du -sh /var/log/*
```

Breaking down:

du - Disk usage
-sh - Summarize, human readable
/var/log/* - All in log directory

Step 3 - Clear logs:

```
sudo journalctl --vacuum=100M
```

Breaking down:

sudo journalctl - System logs
--vacuum=100M - Delete old logs, keep 100MB

Step 4 - Find oldest files:

```
find /var/log -type f -mtime +30
```

Breaking down:

```
find - Find files
/var/log - Directory
-type f = files only
-mtime +30 = modified more than 30 days ago
```

Delete them:

```
find /var/log -type f -mtime +30 -delete
```

Breaking down:

```
-delete = remove files
```

Scenario 3: SSL Certificate Error

Problem: "SSL certificate expired"

Check certificate expiration:

```
echo | openssl s_client -servername example.com -connect
example.com:443 2>/dev/null | grep -E "notAfter|notBefore"
```

Breaking down:

```
openssl s_client - Connect and show certificate
-servername example.com - SNI hostname
-connect example.com:443 - Connect to
2>/dev/null - Hide errors
| grep -E "notAfter|notBefore" - Show dates
```

Get certificate path:

sudo find /etc -name "*.crt" 2>/dev/null

Breaking down:

find - Find files
/etc - Directory
-name "*.crt" = certificate files
2>/dev/null - Hide permission errors

SEARCHING FOR HELP

Man pages:

man service_name

Breaking down:

man - Manual pages
service_name - What to read about

Search man pages:

man -k "keyword"

Breaking down:

man - Manual
-k = search keyword

Get command help:

command --help

Breaking down:

command - Any command
--help = long form help flag

or

command -h

Breaking down:

-h = short form help flag

Internet search strategy:

When stuck, search exactly:

"error message in quotes" + what you're doing
example: "SSL certificate verify failed" nginx

Include version:

"error message" nginx 1.18

Include OS:

"error message" ubuntu 22.04 nginx

COMMON TROUBLESHOOTING MISTAKES

Mistake 1: Jumping to Solutions

Problem appears, you apply first fix you find.

What you should do:

1. Observe - Run top, check logs, see what's happening
2. Understand - What changed? When did it start?
3. Isolate - What's the actual problem, not a symptom?
4. Then fix

Mistake 2: Not Checking Logs

You restart service, hope it works.

Always check:

journalctl -u service_name -n 50

Breaking down:

journalctl -u service_name - Service logs
-n 50 = last 50 entries

See what's actually failing.

Mistake 3: Assuming Network is Problem

Connection slow = always assume network

Actually check:

1. Local system: ps aux, top, df -h
2. Service: systemctl status, logs
3. Then network: ping, traceroute, ss

Mistake 4: Changing Too Much at Once

You change 5 things, one works, don't know which.

Change one thing at a time. Test after each change.

Mistake 5: Not Documenting

You fix it, forget how, breaks again.

Document:

What was the problem?
What did you change?
Why did that fix it?
How to prevent next time?

LEVEL 6: CONFIGURATION MANAGEMENT - SET IT UP RIGHT

You can now fix broken systems (Level 5).

But why do they break so much?

Because they're set up wrong from the start.

Level 6 teaches you to set up systems that don't break. To understand how services are configured, how they start, how they manage resources.

This is where you move from reactive (fixing problems) to proactive (preventing them).

SYSTEMD UNITS - HOW SERVICES WORK

Services are defined in unit files. Located in:

/etc/systemd/system/
/lib/systemd/system/

Breaking down:

/etc/systemd/system/ = custom unit files (yours)
/lib/systemd/system/ = system unit files (distribution)

/etc takes precedence over /lib.

View unit file:

cat /lib/systemd/system/nginx.service

Breaking down:

cat - Display file
/lib/systemd/system/nginx.service - Unit file for nginx

Structure of unit file:

[Unit]
Description=Nginx Web Server
After=network.target

[Service]
Type=notify
ExecStart=/usr/sbin/nginx -g daemon off;
ExecReload=/bin/kill -s HUP $MAINPID
Restart=on-failure

[Install]
WantedBy=multi-user.target

Breaking down sections:

[Unit]
Description = what service does
After = start after these units

[Service]
Type = how service runs (simple, notify, forking, etc.)
ExecStart = command to start service
ExecReload = command to reload without stopping
Restart = when to restart (no, always, on-failure, etc.)

[Install]
WantedBy = when to autostart

Create custom unit file:

sudo nano /etc/systemd/system/myapp.service

Breaking down:

sudo nano - Edit as root
/etc/systemd/system/myapp.service - Custom unit file

Example:

[Unit]
Description=My Application
After=network.target

[Service]
Type=simple
User=appuser
ExecStart=/usr/local/bin/myapp
Restart=always

RestartSec=5

[Install]
WantedBy=multi-user.target

Breaking down:

User=appuser - Run as this user (security)
Restart=always - Always restart if stops
RestartSec=5 - Wait 5 seconds between restarts

Enable and start:

sudo systemctl enable myapp.service

Breaking down:

sudo systemctl enable - Enable autostart

sudo systemctl start myapp.service

Breaking down:

sudo systemctl start - Start now

ENVIRONMENT VARIABLES AND CONFIG FILES

Set environment in unit file:

[Service]
Environment="DATABASE_URL=postgres://localhost/mydb"
Environment="LOG_LEVEL=INFO"

ExecStart=/usr/local/bin/myapp

Breaking down:

Environment="KEY=value" - Set env vars for service

Source from file:

[Service]
EnvironmentFile=/etc/myapp/config
ExecStart=/usr/local/bin/myapp

Breaking down:

EnvironmentFile=/etc/myapp/config - Load config file

Config file example (/etc/myapp/config):

DATABASE_URL=postgres://localhost/mydb

LOG_LEVEL=INFO

PORT=4000

Usage in app:

bash:
```
echo $DATABASE_URL
```

python:
```
import os
db_url = os.environ['DATABASE_URL']
```

MANAGING CONFIGURATION FILES

System config locations:

/etc/ = all system configuration
/etc/nginx/ = nginx config
/etc/postgresql/ = PostgreSQL config
/etc/ssh/ = SSH config
/home/user/.config/ = user config

Edit config:

sudo nano /etc/nginx/nginx.conf

Breaking down:

sudo nano - Edit as root (configs need root access)
/etc/nginx/nginx.conf - Nginx config file

Backup before editing:

sudo cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.bak

Breaking down:

cp - Copy
/etc/nginx/nginx.conf - Original file
/etc/nginx/nginx.conf.bak - Backup copy

Test config before reload:

sudo nginx -t

Breaking down:

sudo nginx - Nginx command

-t = test flag. -t = syntax test (don't run)

Shows if config is valid before applying.

Reload after config change:

sudo systemctl reload nginx

Breaking down:

sudo systemctl reload - Reload config (graceful)
nginx - Service

Don't restart, reload. Restart kills connections.

RESOURCE LIMITS AND ISOLATION

Limit memory for service:

[Service]
MemoryLimit=512M
ExecStart=/usr/local/bin/myapp

Breaking down:

MemoryLimit=512M - Max 512MB RAM for service

Limit CPU:

[Service]
CPUQuota=50%
ExecStart=/usr/local/bin/myapp

Breaking down:

CPUQuota=50% - Max 50% of one CPU core

Limit disk I/O:

```
[Service]
IOWeight=100
ExecStart=/usr/local/bin/myapp
```

Breaking down:

IOWeight=100 - Disk I/O weight (lower = less I/O)

Check limits:

```
systemctl show myapp.service | grep Limit
```

Breaking down:

systemctl show myapp.service - Show unit properties
| grep Limit - Filter for limits

SERVICE DEPENDENCIES

Start order with After:

```
[Unit]
Description=Web App
After=postgresql.service mysql.service
```

Breaking down:

After = postgresql.service - Start after PostgreSQL
mysql.service - And after MySQL

Require dependency:

[Unit]
Description = Web App
Requires = postgresql.service

Breaking down:

Requires = postgresql.service - Must have PostgreSQL running

If PostgreSQL stops, web app stops too.

Weaker dependency:

[Unit]
Wants = postgresql.service

Breaking down:

Wants = postgresql.service - Try to start PostgreSQL
But don't fail if not available

SOCKET ACTIVATION

Lazy service startup (only start when needed):

Create socket file: /etc/systemd/system/myapp.socket

```
[Unit]
Description=My App Socket
Before=myapp.service

[Socket]
ListenStream=5000
Accept=no

[Install]
WantedBy=sockets.target
```

Breaking down:

```
[Socket]
ListenStream=5000 - Listen on port 5000
Accept=no - Single instance
```

Create service file to use socket:

```
[Unit]
Description=My App
Requires=myapp.socket

[Service]
Type=notify
ExecStart=/usr/local/bin/myapp
StandardInput=socket
```

Breaking down:

Requires=myapp.socket - Depend on socket
StandardInput=socket - Accept connections from

socket

Enable socket:

sudo systemctl enable myapp.socket

Breaking down:

sudo systemctl enable - Enable socket autostart

TIMER UNITS - SCHEDULED SERVICES

Create timer: /etc/systemd/system/backup.timer

[Unit]
Description=Daily Backup Timer
Requires=backup.service

[Timer]
OnCalendar=daily
OnCalendar=00:02

[Install]
WantedBy=timers.target

Breaking down:

[Timer]
OnCalendar=daily - Run every day
OnCalendar=00:02 - At 00:02 AM

Create service: /etc/systemd/system/backup.service

```
[Unit]
Description=Backup Service

[Service]
Type=oneshot
ExecStart=/usr/local/bin/backup.sh
```

Breaking down:

Type=oneshot - Run once and exit

Enable timer:

sudo systemctl enable backup.timer

Breaking down:

sudo systemctl enable - Enable autostart

List timers:

sudo systemctl list-timers

Breaking down:

systemctl list-timers - Show active timers

LOGGING CONFIGURATION

Send service logs to syslog:

```
[Service]
StandardOutput=journal
```

```
StandardError=journal
SyslogIdentifier=myapp
```

Breaking down:

StandardOutput=journal - Send stdout to systemd journal
StandardError=journal - Send stderr to systemd journal
SyslogIdentifier=myapp - Log identifier

View logs:

```
journalctl -u myapp
```

Breaking down:

journalctl -u myapp - Logs for myapp service

Log verbosity:

```
[Service]
Environment="LOG_LEVEL=DEBUG"
ExecStart=/usr/local/bin/myapp
```

Breaking down:

LOG_LEVEL=DEBUG - Set log level (varies by app)

REAL-WORLD CONFIGURATION PATTERNS

Pattern 1: Web Application with Database

Create: /etc/systemd/system/webapp.service

```
[Unit]
Description=Web Application
After=postgresql.service

[Service]
Type=simple
User=webuser
ExecStart=/usr/local/bin/webapp
EnvironmentFile=/etc/webapp/config
Restart=on-failure
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Breaking down:

After=postgresql.service - Start after database
User=webuser - Run as unprivileged user
EnvironmentFile=/etc/webapp/config - Load config

Pattern 2: Long-Running Process with Restarts

```
[Service]
Type=simple
ExecStart=/usr/local/bin/monitor
Restart=always
RestartSec=5
StandardOutput=journal
StandardError=journal
```

Breaking down:

Restart=always - Always restart if crashes
RestartSec=5 - Wait 5 seconds between restarts

Pattern 3: CPU-Bound Service with Limits

```
[Service]
ExecStart=/usr/local/bin/processor
CPUQuota=75%
CPUAffinity=0-3
IOWeight=50
```

Breaking down:

CPUQuota=75% - Max 75% CPU
CPUAffinity=0-3 - Use cores 0-3 only
IOWeight=50 - Limited disk I/O

Pattern 4: Daily Maintenance Task

Create timer:

```
[Unit]
Description=Maintenance Timer

[Timer]
OnCalendar=*-*-* 02:30:00
Persistent=true

[Install]
WantedBy=timers.target
```

Breaking down:

OnCalendar=*-*-* 02:30:00 = Every day at 2:30

AM
Persistent=true = Run if system was off at scheduled time

COMMON CONFIGURATION MISTAKES

Mistake 1: Not Testing Config

You edit config, restart, service broken.

Always test first:

sudo nginx -t

Breaking down:

nginx -t = syntax test (service specific, may vary)

Most services have test flags.

Mistake 2: Using daemon off/foreground Wrong

Service must run in foreground for systemd:

ExecStart=/usr/sbin/nginx -g daemon off;

Breaking down:

daemon off = run in foreground (required for systemd)

If daemon mode on, systemd thinks service stopped.

Mistake 3: Not Setting User

Service runs as root by default. Security risk:

```
[Service]
User=appuser
Group=appgroup
```

Breaking down:

```
User=appuser  =  unprivileged user
Group=appgroup  =  group
```

Mistake 4: No Restart Policy

Service crashes, stays dead:

```
[Service]
Restart=on-failure
RestartSec=10
```

Breaking down:

```
Restart=on-failure  =  restart if exit code is error
RestartSec=10  =  wait 10 seconds between restarts
```

Mistake 5: Blocking Calls in ExecStart

ExecStart=/usr/local/bin/very-slow-startup

If startup takes forever, systemd might kill it.

Use Type=notify for long startups:

```
[Service]
Type=notify
ExecStart=/usr/local/bin/long-startup
```

Breaking down:

Type=notify = service must signal ready (see docs)

LEVEL 7: SECURITY - PROTECT WHAT MATTERS

You've built stable systems.

But are they safe?

Security isn't added at the end. It's built from the beginning.

Level 7 teaches you to think like an attacker, then defend against attacks.

USERS AND GROUPS - ACCESS CONTROL

Create user:

sudo adduser appuser

Breaking down:

sudo adduser - Create user
appuser = username

Creates home directory, sets up shell.

Add user to group:

sudo usermod -aG sudo appuser

Breaking down:

sudo usermod - Modify user
-a = append. -a = add to existing groups (don't remove)
-G sudo = group. -G = supplementary groups. sudo = group name
appuser = username

Make user a sudoer (admin):

sudo usermod -aG sudo appuser

Breaking down:

-aG sudo = append to sudo group

Remove user:

sudo deluser appuser

Breaking down:

sudo deluser - Delete user
appuser = username

Delete with home:

sudo deluser --remove-home appuser

Breaking down:

--remove-home = delete home directory too

List users:

cat /etc/passwd

Breaking down:

cat /etc/passwd - User database file (human-readable)

List groups:

cat /etc/group

Breaking down:

cat /etc/group - Group database file

FILE PERMISSIONS - CONTROLLING ACCESS

Octal permissions:

7 = 111 = rwx (read, write, execute)
6 = 110 = rw- (read, write)
5 = 101 = r-x (read, execute)
4 = 100 = r-- (read)
0 = 000 = --- (no access)

Set permissions:

chmod 600 ~/.ssh/id_rsa

Breaking down:

chmod 600 - Set permissions
6 = owner (read, write)
0 = group (none)

0 = others (none)
~/.ssh/id_rsa = private key (must be 600)

Public key permissions:

chmod 644 ~/.ssh/id_rsa.pub

Breaking down:

644 = rw-r--r-- (owner can read/write, others read only)

Directory permissions:

chmod 755 /var/www

Breaking down:

755 = rwxr-xr-x
7 = owner full access
5 = group/others can read and enter

Recursive permissions:

chmod -R 755 /var/www

Breaking down:

-R = recursive (all files and subdirectories)

Special permissions:

Setuid (run as owner):

chmod u+s /usr/bin/sudo

Breaking down:

u+s = setuid bit. u = user. + = add. s = setuid

Setgid (run as group):

chmod g+s /usr/local/bin/setgid_prog

Breaking down:

g+s = setgid bit. g = group. + = add. s = setgid

Sticky bit (only owner can delete):

chmod +t /tmp

Breaking down:

+t = sticky bit. t = sticky. Everyone can create but not delete others' files

SUDOERS - PRIVILEGE ESCALATION

Edit sudoers file safely:

sudo visudo

Breaking down:

sudo visudo - Edit sudoers with syntax check
(must use visudo, not nano, to prevent locking yourself out)

Grant user sudo access:

appuser ALL=(ALL) ALL

Breaking down:

appuser = which user
ALL = on all hosts
(ALL) = can run as any user
ALL = all commands

More restrictive:

appuser ALL=(root) /usr/bin/apt, /usr/bin/systemctl

Breaking down:

appuser = which user
(root) = only as root
/usr/bin/apt, /usr/bin/systemctl = only these commands

No password:

appuser ALL=(ALL) NOPASSWD: ALL

Breaking down:

NOPASSWD: = don't ask for password (for scripts)

Allow specific command:

appuser ALL=(ALL) NOPASSWD: /usr/bin/systemctl restart nginx

Breaking down:

NOPASSWD: /usr/bin/systemctl restart nginx = just this command, no password

SSH HARDENING

Disable password authentication:

sudo nano /etc/ssh/sshd_config

Breaking down:

sudo nano - Edit as root
/etc/ssh/sshd_config = SSH server config

Find and set:

PasswordAuthentication no
PubkeyAuthentication yes
PermitRootLogin no
Port 2222

Breaking down:

PasswordAuthentication no = keys only, no passwords
PubkeyAuthentication yes = allow key auth
PermitRootLogin no = cannot SSH as root
Port 2222 = non-standard port (security by obscurity)

Restart SSH:

sudo systemctl reload sshd

Breaking down:

systemctl reload sshd = apply changes (graceful)

Key-only configuration:

sudo nano /etc/ssh/sshd_config

Breaking down:

Set these:

PasswordAuthentication no
PubkeyAuthentication yes
PermitEmptyPasswords no
AllowUsers user1 user2

Breaking down:

AllowUsers user1 user2 = only these users can SSH

FIREWALL HARDENING

Default deny everything:

sudo ufw default deny incoming
sudo ufw default allow outgoing

Breaking down:

default deny incoming = block everything in
default allow outgoing = allow everything out

Explicitly allow services:

```
sudo ufw allow 22/tcp
sudo ufw allow 80/tcp
sudo ufw allow 443/tcp
```

Breaking down:

allow 22/tcp = SSH (TCP protocol)
allow 80/tcp = HTTP
allow 443/tcp = HTTPS

Allow from specific IP:

```
sudo ufw allow from EXAMPLE_IP
```

Breaking down:

allow from EXAMPLE_IP = only from this IP

Rate limit SSH:

```
sudo ufw limit 22/tcp
```

Breaking down:

limit 22/tcp = rate limit (blocks brute force attempts)

Enable firewall:

```
sudo ufw enable
```

Breaking down:

enable = turn on firewall

UPDATES AND PATCHING

Check available updates:

sudo apt update
sudo apt list --upgradable

Breaking down:

apt update = refresh package lists
apt list --upgradable = show what can be upgraded

Install updates:

sudo apt upgrade

Breaking down:

apt upgrade = install updates (safe)

or for kernel updates:

sudo apt full-upgrade

Breaking down:

apt full-upgrade = install everything including kernel (less safe)

Install security updates automatically:

sudo apt install unattended-upgrades

Breaking down:

unattended-upgrades = auto-install security updates

Configure:

sudo nano /etc/apt/apt.conf.d/50unattended-upgrades

Breaking down:

Configuration file for auto-updates

COMMON SECURITY MISTAKES

Mistake 1: Running as Root

Services run as root unnecessarily:

```
[Service]
User=appuser
```

Breaking down:

User=appuser = unprivileged user (if service doesn't need root)

If service compromised, attacker only gets appuser permissions.

Mistake 2: World-Readable Secrets

Passwords in world-readable files:

ls -la config.txt

Returns: -rw-r--r--

Should be:

chmod 600 config.txt

Breaking down:

chmod 600 = only owner can read

Mistake 3: Default SSH Port

Bots scan port 22. Move it:

Port 2222

Breaking down:

Port 2222 = non-standard port (reduces noise, not real security)

Mistake 4: Old Packages

Unpatched vulnerabilities:

sudo apt update
sudo apt upgrade

Breaking down:

Must run regularly to get security updates

Mistake 5: No Backups

System compromised, no backups:

Regular backups:

sudo backup-tool

Breaking down:

Automate with cron or systemd timer

LEVEL 8: PERFORMANCE - MAKE SYSTEMS FAST

You've built stable, secure systems.

But are they fast?

Level 8 teaches you to identify bottlenecks and fix them.

IDENTIFYING BOTTLENECKS

Start with uptime:

uptime

Breaking down:

uptime - Shows load average

Load 1.0 = one core fully used. On 4-core: 4.0 = fully loaded.

Check CPU usage:

```
top -b -n 1 | head -20
```

Breaking down:

top - Process monitor
-b = batch (output once)
-n 1 = run once
| head -20 = first 20 lines

Shows % CPU by process.

Check memory:

```
free -h
```

Breaking down:

free - Memory statistics
-h = human readable

Shows used/free memory.

Check disk:

```
df -h
```

Breaking down:

df - Disk free
-h = human readable

Check disk I/O:

```
iostat -x 1 5
```

Breaking down:

iostat - IO statistics
-x = extended
1 = 1 second interval
5 = 5 samples

Shows disk usage and wait time.

Check network:

iftop

Breaking down:

iftop - Interface top (bandwidth usage)

Shows which hosts using bandwidth.

OPTIMIZING CPU USAGE

Find CPU hog:

top

Breaking down:

top - Process monitor

Sort by CPU %

If single threaded, use multiple workers:

Nginx:

worker_processes auto;

Breaking down:

auto = use number of cores

Django/Gunicorn:

gunicorn --workers 4 app.py

Breaking down:

--workers 4 = 4 worker processes

Python multiprocessing:

from multiprocessing import cpu_count
workers = cpu_count()

Breaking down:

cpu_count() = get number of cores

Use efficient algorithms:

Profile code:

python -m cProfile script.py

Breaking down:

cProfile - Python profiler
script.py = script to profile

Shows which functions use most CPU.

OPTIMIZING MEMORY

Find memory hog:

top

Breaking down:

top - Shows RSS (memory usage)

Sort by memory %

Reduce allocations:

Avoid creating objects in loops:

```
# Bad:
for i in range(1000000):
 obj = expensive_object()

# Good:
obj = expensive_object()
for i in range(1000000):
 obj.reset()
```

Breaking down:

Reuse objects instead of recreating

Cache results:

```
from functools import lru_cache

@lru_cache(maxsize=128)
```

```
def expensive_function(x):
 # expensive computation
```

Breaking down:

lru_cache = least recently used cache
Cache results to avoid recomputation

Monitor for leaks:

```
ps aux | grep process
```

Breaking down:

Watch RSS column. If increasing, memory leak.

OPTIMIZING DISK I/O

Batch I/O:

```
# Bad:
for file in files:
 process_and_write(file)

# Good:
batch = []
for file in files:
 batch.append(process(file))
write_all(batch)
```

Breaking down:

Batch operations to reduce write calls

Use SSD cache:

Most systems now use SSDs (fast).

Enable caching:

Disable unnecessary syncs:

In database:

sync_binlog = 1000

Breaking down:

Batch syncs instead of per transaction

OPTIMIZING NETWORK

Compression:

Nginx:

```
gzip on;
gzip_min_length 1000;
```

Breaking down:

gzip on = enable compression
gzip_min_length 1000 = compress if > 1000 bytes

Caching:

Nginx:

```
proxy_cache_path /var/cache/nginx levels=1:2
keys_zone=my_cache:10m;
```

Breaking down:

```
proxy_cache_path = cache location
keys_zone=my_cache:10m = 10MB cache
```

Connection pooling:

Don't create new connections per request.

Reuse persistent connections.

DATABASE PERFORMANCE

Check slow queries:

MySQL:

```
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 2;
```

Breaking down:

```
slow_query_log = enable
long_query_time = 2 seconds = slow query threshold
```

Add indexes:

```
ALTER TABLE users ADD INDEX idx_email (email);
```

Breaking down:

ADD INDEX idx_email (email) = index on email column

Avoid N+1 queries:

```
# Bad:
for user in users:
 posts = get_posts(user.id)  # Query per user

# Good:
users_with_posts = query(
 "SELECT u.*, p.* FROM users u LEFT JOIN posts p ON u.id =
p.user_id"
)
```

Breaking down:

One query instead of N queries

PROFILING AND MONITORING

System profiling:

```
perf record ./program
```

Breaking down:

perf - Performance analysis tool
record - Record execution

```
perf report
```

Breaking down:

perf report - Analyze recording

Shows where time is spent.

Application monitoring:

Prometheus + Grafana (standard monitoring stack)

Or New Relic, DataDog, etc.

Track:

CPU usage
Memory usage
Disk I/O
Network
Response times
Error rates

COMMON PERFORMANCE MISTAKES

Mistake 1: Optimizing Wrong Thing

You optimize CPU when bottleneck is disk I/O.

Always measure first:

top, iostat, free, df

Breaking down:

Find actual bottleneck before optimizing

Mistake 2: Premature Optimization

Optimize code before it's slow.

Write clear code first. Optimize slow parts.

Mistake 3: Not Caching

Every request hits database.

Add caching:

Redis
Memcached
Application-level cache

Mistake 4: Bad Database Queries

SELECT * queries without indexes.

Use EXPLAIN:

EXPLAIN SELECT * FROM big_table WHERE email = 'test@example.com';

Breaking down:

EXPLAIN = show query plan (will use index?)

Add indexes if needed:

ALTER TABLE big_table ADD INDEX idx_email (email);

Mistake 5: No Monitoring

System slow, don't know why.

Set up monitoring:

Prometheus, Grafana, or similar

Watch:

CPU
Memory
Disk
Network

YOUR LINUX MASTERY JOURNEY - COMPLETE

You came here not knowing how Linux worked.

You're leaving as someone who:

- Understands the system deeply
- Can build stable, secure systems
- Can troubleshoot and fix problems
- Can automate complex tasks
- Can optimize for performance
- Can explain every command and flag

This is not the end. This is the beginning.

With Linux mastery, you can:

- Deploy web services that millions use
- Manage infrastructure at scale
- Understand the foundation of modern computing
- Build systems that last

The tools you've learned are used in:

- Every cloud service (AWS, Azure, GCP)
- Every Kubernetes cluster
- Every Docker container
- Every production system

The knowledge here is worth more than computer science degrees to employers.

You have mastered the operating system that powers the internet.

What comes next is up to you.